Un Servidor de Aplicaciones MLS: Especificación y Verificación de Propiedades de Seguridad

Abel Valente

27 de mayo de 2003

Índice General

Índice General iii

Abstract vii

Agradecimientos ix

1 Introducción 1
- 1.1 Técnicas formales aplicadas a seguridad . . . 2
- 1.2 Objetivo de este trabajo . . . 2
- 1.3 Organización . . . 3

2 Control de Acceso 5
- 2.1 Seguridad en Tecnología de la Información . . . 6
  - 2.1.1 Confidencialidad . . . 6
  - 2.1.2 Control de Acceso . . . 7
- 2.2 Control de Acceso Discrecionario . . . 7
- 2.3 Control de Acceso Obligatorio . . . 9
  - 2.3.1 Política de seguridad del Departamento de Defensa . . . 9
  - 2.3.2 Modelos de Seguridad Multi-Nivel . . . 11
- 2.4 Caballos de Troya . . . 14
- 2.5 DAC, MAC y Caballos de Troya . . . 15

3 Bell-LaPadula 19
- 3.1 Contenido de los artículos . . . 20
- 3.2 Modelo de sistema . . . 20
  - 3.2.1 Modos de acceso . . . 20
  - 3.2.2 Requerimientos y Decisiones . . . 21
- 3.3 Propiedades de Seguridad . . . 22
  - 3.3.1 Clases de Seguridad . . . 23
  - 3.3.2 Seguridad Simple . . . 23
  - 3.3.3 Propiedad-★ . . . 24
- 3.4 Aplicando BLP . . . 24
  - 3.4.1 Reglas de Operación . . . 25
  - 3.4.2 Verificación de las Propiedades de Seguridad . . . 27
- 3.5 Conclusión . . . 28

4 DAC+MLS en un Servidor de Aplicaciones **29**
4.1 Servidores de Aplicaciones 30
4.1.1 Seguridad en un Servidor de Aplicaciones 30
4.1.2 Arquitecturas de Control de Acceso 32
4.2 El Módulo de Control de Acceso 34
4.2.1 Funcionalidad del módulo 34
4.2.2 Responsabilidades de la aplicación cliente 35
4.3 Conclusión 36

5 Especificación **37**
5.1 Objetivo y estructura del capítulo 38
5.2 Acerca de la especificación y Z 39
5.3 Especificación del estado del módulo 40
5.3.1 Objetos y Sujetos 40
5.3.2 Derechos y Permisos de Acceso 41
5.3.3 Permisos de Acceso vigentes 44
5.3.4 Derechos de Acceso 46
5.3.5 Clases de Seguridad 47
5.3.6 Adminstración de MAC 48
5.3.7 Estado completo del módulo 49
5.3.8 Estado Inicial 49
5.4 Especificación de la política de seguridad 50
5.4.1 La relaciones *dominates* y *matches* 51
5.4.2 La relación *secureRel* 51
5.4.3 Propiedad de Seguridad Simple 52
5.4.4 Propiedad-★ 53
5.5 Especificación de la operaciones 54
5.5.1 Interfaz del módulo 54
5.5.2 Definiciones auxiliares 56
5.5.3 GetRead 60
5.5.4 GetWrite 64
5.5.5 GetExecute 65
5.5.6 GetReadWrite 66
5.5.7 Release 67
5.5.8 Give y Rescind 68
5.5.9 ChangeObjClass 68
5.5.10 CreateObject y DeleteObject 69
5.6 Preservación de la Política de Seguridad 70
5.6.1 Propiedad de Seguridad Simple 70
5.6.2 Propiedad-★ 71
5.6.3 Conjunción de las propiedades 71

6 Verificación **73**
6.1 Mecanismo de prueba de Z/EVES 74
6.2 Lemas de BLP 74
6.3 Estrategia para la prueba de consistencia 75
6.3.1 Factorización de la prueba 75
6.3.2 Casos derivados del consecuente 75
6.3.3 Casos derivados del antecedente 76
6.4 Teoremas auxiliares para casos de rechazo 76
6.4.1 Propiedad de Seguridad Simple 76
6.4.2 Propiedad-★ 78
6.5 Teoremas auxiliares para los casos válidos 81
6.5.1 Teoremas para *augb* 82
6.5.2 Teoremas para *objSAtts* en su interacción con *augb* 83
6.5.3 Teoremas del artículo de BLP 84
6.5.4 Propiedad de Seguridad Simple en un caso válido 85
6.5.5 Propiedad-★ en un caso válido 87
6.6 Certificación de GetRead 92
6.6.1 En un caso de rechazo 93
6.6.2 En un caso válido 93
6.6.3 Consistencia total 94
6.7 Pruebas de chequeo de dominio 95

7 Aplicación **97**
7.1 Un caso de uso 98
7.2 Análisis de preservación de seguridad 99
7.2.1 Especificación 99
7.2.2 Verificación 100

8 Conclusión **103**

A Designaciones **105**

B Especificación Completa **109**

C Pruebas Completas para Z/EVES 2.1 **123**

Bibliografía **151**

Abstract

Se especificará formalmente un modelo de seguridad multi-nivel, más concretamente, el modelo Bell-LaPadula, en el contexto de un módulo de control de acceso para un servidor de aplicaciones. Se implementará, además, un mecanismo de control de acceso discrecionario basado en Listas de Control de Acceso (*ACL*). El objetivo de la especificación es la posterior prueba formal de algunas propiedades de seguridad del módulo. La especificación se escribirá en Z, y para las pruebas se utilizará el asistente de pruebas Z/EVES 2.1. Con éste se verificarán algunas propiedades de la especificación, y en particular se probará que una de las operaciones del módulo preserva las propiedades de seguridad deseadas.

Agradecimientos

(*en orden alfabético*)

Victor Martín Dias
Hernán Badenes
Gabriel Baum
Esteban de la Canal
Fernanda Larrain
Pablo E. Martínez López
Claudia Pons
Patricio Reyna Almandos
Luciana Valente

Y especialmente a *Maximiliano Cristiá*,
por su invalorable apoyo y dedicación.

Capítulo 1

Introducción

La seguridad en los sistemas de cómputo modernos se ha vuelto una cualidad esencial. Una de las causas del incremento de su importancia es la creciente informatización de servicios críticos, tales como sumistros básicos de infraestructura nacional (electricidad o gas, por ejemplo), o como las telecomunicaciones, redes de transporte o servicios de salud. La vastedad de Internet es una arma de doble filo, a la vez ofreciéndonos muchos servicios o comodidades y por otro lado una enorme cantidad de amenazas. Día a día se ve como usuarios hogareños, en especial los que disponen de conexiones permanentes, optan por instalar uno o más programas de protección (*Firewalls*, etc.) ante la amenaza de perder datos personales o laborales. Siguiendo esta línea, uno podría preguntarse cuál es la perspectiva para los casos cruciales, como por ejemplo, un hospital. Todas las grandes instituciones poseen un portal de servicios a través de Internet, lo que les da indefectiblemente una vulnerabilidad aumentada.

Es claro que este tipo de sistemas de cómputo, por su naturaleza, *demandan* confiabilidad a la hora de evitar adversidades tanto ocasionales como intencionales. Es necesario llevar a cabo un análisis profundo de los escenarios que comprometen la seguridad ya desde la etapa de diseño. Cerciorarse de que los sistemas diseñados *toleren* tanto ataques previstos como, en lo posible, ataques imprevistos, deja de ser un objetivo de segundo plano. La naturaleza de la mayoría de los ataques a la seguridad se centra en encontrar debilidades accidentales, por lo que la única manera de controlarlos efectivamente es mediante una implementación cuidadosa y un diseño fuerte. Es más efectivo concentrarse a nivel de diseño en desarrollar una buena metodología para el control de las irrupciones ilegales que verificar cada pequeña porción de programa en busca de un defecto. Un buen diseño debe reducir las oportunidades de aprovechar los accidentes de implementación.

1.1 Técnicas formales aplicadas a seguridad

Creciente es la demanda de métodos seguros para realizar transacciones, brindar servicios y mantener relaciones financieras en la WWW [10]. Sin embargo, los problemas de seguridad, si finalmente son tenidos en cuenta, se suelen manejar de manera informal. Esto es sorprendente, pues numerosos inconvenientes causados por ataques de diversos tipos podrían haberse evitado o disminuido si el problema se hubiera especificado y verificado con técnicas formales. Las técnicas formales nos ofrecen la posibilidad de *probar* que tales amenazas son toleradas o, en el caso de una irrupción, que no pueden causar graves daños. Un modelo de seguridad diseñado o implementado sin un modelo formal carece del rigor matemático necesario para abordar el problema.

Los modelos formales para seguridad nacieron para servir al Departamento de Defensa de los Estados Unidos con el modelo Bell-LaPadula [9, 8], el cual sigue aún en uso junto con una variedad de nuevos modelos. Todos ellos sirven para probar que, bajo una determinada política, un sistema es seguro. Los modelos de no-interferencia, no-deducibilidad y BIBA, son otros ejemplos [2, 3, 11].

Este trabajo presenta el diseño de un módulo que ofrece servicios para control de acceso. La ingeniería utilizada, constituida por un lenguaje de especificación formal muy instituido, una herramienta de asistencia de pruebas sobre este lenguaje y la base de un modelo matemático para seguridad informática muy estudiado, muestra la aplicabilidad de las técnicas elegidas para encarar este tipo de soluciones.

1.2 Objetivo de este trabajo

El objetivo de este trabajo es especificar y verificar un modelo de seguridad multi-nivel, más concretamente, el modelo Bell-LaPadula. El contexto para dicha especificación es el módulo de control de acceso de un servidor de aplicaciones, donde se conjugará con un mecanismo discrecionario basado en Listas de Control de Acceso. Debido a que está orientado a ofrecer sus servicios a clientes de funcionalidad crítica, se especificarán formalmente todas las funciones y se verificará que una de ellas preserva la política de seguridad. La especificación será escrita en Z [7, 15] y verificada usando el asistente de pruebas Z/EVES.

Se mostrará la especificación y verificación de los componentes más importantes del módulo de control de acceso. Así, este ofrecerá una base para la construcción de aplicaciones críticas o semi-críticas en cuanto a divulgación de la información contenida.

1.3 Organización

Capítulos

El presente trabajo está organizado a lo largo de los capítulos que siguen de esta manera:

Capítulo 2 (p.5), Control de Acceso · Presenta un resumen de los tópicos de seguridad necesarios para abordar los capítulos posteriores.

Capítulo 3 (p.19), Bell-LaPadula · Introduce de manera abreviada el modelo que constituye la base de la especificación del módulo, Bell-LaPadula.

Capítulo 4 (p.29), DAC+MLS en un Servidor de Aplicaciones · Explica el escenario en el cual se quiere diseñar el módulo de control de acceso, la funcionalidad deseada del mismo, y sus relaciones arquitectónicas.

Capítulo 5 (p.37), Especificación · Presenta la especificación formal del módulo de control de acceso y de su política de seguridad.

Capítulo 6 (p.73), Verificación · Presenta las pruebas realizadas con el asistente de pruebas Z/EVES de las propiedades de la especificación.

Capítulo 7 (p.97), Aplicación · Explica la manera de aplicar la especificación al desarrollo de aplicaciones clientes del módulo.

Capítulo 8 (p.103), Conclusión · Resume las conclusiones de este trabajo.

Apéndices

Se presentarán, además, los siguientes apéndices, que exhiben todos los detalles omitidos entre líneas a lo largo de los capítulos:

Apéndice A (p.105), Designaciones · Presenta todas las designaciones de las entidades definidas en la especificación.

Apéndice B (p.109), Especificación Completa · Presenta la versión completa de la especificación. Incluye, aunque sin comentar, lo presentado en el capítulo 5 así como también todos los detalles omitidos.

Apéndice C (p.123), Pruebas Completas para Z/EVES 2.1 · Enumera los teoremas probados en este trabajo. Incluye tanto las incluidas en el capítulo 6 como las omitidas, todas sin comentar.

Capítulo 2

Control de Acceso

En este capítulo introduciremos algunos conceptos de seguridad informática y explicaremos las bases para entender los modelos para control de acceso que utilizaremos en la especificación: Listas de Control de Acceso y Bell-LaPadula.

Las Listas de Control de Acceso son un tipo de control de acceso discrecionario, el esquema más conocido. Por otro lado, Bell-LaPadula es un modelo de control de acceso con una política obligatoria, al que dedicaremos la mayor parte del esfuerzo de nuestro trabajo. Las siguientes secciones tienen por objetivo explicar estos conceptos. Mostraremos las ventajas de cada esquema y las razones por las que se quiere que ambos modelos participen en la política de seguridad del módulo de control de acceso.

2.1 Seguridad en Tecnología de la Información

Seguridad es una práctica por la cual individuos y organizaciones protegen sus propiedades físicas e intelectuales contra toda forma de ataque.

Aunque los problemas de seguridad no son nuevos, hay un interés renovado en el área de seguridad en sistemas de cómputo. Esto se debe a que los sistemas de información actuales se convirtieron en repositorios para pertenencias tanto personales como corporativas. Computadoras de todos los tamaños y configuraciones de red proveen nuevos niveles de acceso y consecuentemente nuevas oportunidades para posibles abusos.

2.1.1 Confidencialidad

Los problemas de seguridad informática se suelen clasificar en tres categorías respecto del tipo de ataque que intentan resistir:

Confidencialidad · Prevenir la divulgación no autorizada de información.

Integridad · Controlar los intentos de alterar información indebidamente.

Disponibilidad · Evitar ataques a la performance del sistema.

Confidencialidad siempre ha sido el área de mayor interés en seguridad informática debido a que las soluciones a este problema suelen ser la base de las soluciones al problema de la *Integridad*. Por otro lado, *Disponibilidad* nunca ha sido un área muy estudiada por la comunidad científica del área. Hay dos razones fundamentales para esto último. La primera es que el gobierno de los Estados Unidos, el principal inversor en estudios para seguridad informática, siempre se ha interesado más en confidencialidad e integridad que en esta última area. La segunda razón es que el problema es fundamentalmente más dificil, ya que involucra directamente la corrección funcional *total* de un sistema de cómputo, cosa extremadamente difícil de lograr.

Por resultar el área de mayor interés elegimos en este trabajo atacar el problema de la confidencialidad de la información. Algunos ejemplos de organizaciones que cuentan con políticas de seguridad orientadas a la confidencialidad son:

- Sistemas bancarios en aquellos estados con leyes de secreto bancario.
- Organizaciones vinculadas con servicios de inteligencia y seguridad.
- Oficinas estatales que deban proteger datos de sus ciudadanos o documentos.

2.1.2 Control de Acceso

Uno de los objetivos principales de los mecanismos de seguridad en un sistema de cómputo es controlar el acceso a la información. No fue hasta temprano en la década los '70 que se advirtió que hay dos tipos de control de acceso fundamentalmente diferentes:

Control de Acceso Discrecionario · Al que referiremos usualmente con *DAC* (*Discretionary Access Control*), es el primero y más común de los tipos de control de acceso. Los usuarios del sistema (*sujetos*) entregan o retiran permisos de acceso a sus archivos (en general, *objetos*).

Control de Acceso Obligatorio · Nos referiremos a este tipo de control de acceso usando la sigla *MAC* (*Mandatory Access Control*). En este tipo de control de acceso un administrador asigna atributos de seguridad a los objetos y sujetos, y luego se basa en estos para decidir si un sujeto puede acceder a un objeto. Estos atributos no pueden ser modificados por los sujetos o sus programas ordinarios.

No hay políticas que no puedan ser clasificadas en alguna de estas dos categorías (también en ambas). Notar que MAC es llamado también "No Discrecionario", ya que puede mostrarse que si una política no es DAC entonces debe ser MAC y viceversa [1].

2.2 Control de Acceso Discrecionario

La mayoría del software de base comercial implementa alguna forma de control de acceso de este tipo. Incluso antes de que las técnicas de control de acceso llegaran a ser implementadas como un mecanismo de protección intrínseco al sistema operativo, se podían ver ejemplos de DAC cuando un operador de cintas cargaba la cinta a pedido de un usuario y verificaba que el mismo tenga los permisos necesarios para utilizarla.

Cuando los sistemas operativos ofrecieron discriminación por usuarios y permitieron algún tipo de almacenamiento de archivos, entonces fue necesario implementar control de acceso en el mismo SO, ya que de otra manera cualquier usuario que tuviera acceso a la computadora tendría libre acceso a todos los archivos de los demás usuarios.

Un mecanismo de control de acceso es *discrecionario* si los usuarios ordinarios pueden participar en la definición de su política. Este es, por ejemplo, el caso de UNIX, dado que cualquier usuario puede definir los atributos de seguridad de los archivos que posee y, por lo tanto, tiene la capacidad de negar o brindar acceso a los datos de su propiedad, alterando la política de seguridad.

Las siguientes son técnicas de control de acceso discrecionario:

Acceso a Archivo por Contraseña · La técnica de *Acceso a Archivo por Contraseña* surgió entre los primeros mecanismos. Los archivos podían tener asociada una contraseña, y quien quisiera accederlos necesitaba conocerla. Como es de esperar, esta primera técnica no es satisfactoria en muchos aspectos. No se le puede quitar el acceso a un sujeto determinado sin afectar a todo el resto (aunque podemos solucionar parcialmente el problema asignando multiples contraseñas por archivo). Además, requiere que los sujetos memoricen demasiados contraseñas, lo cual tiende a ser inapropiado.

Lista de Capacidades · Otra técnica que se usa para implementar DAC es la *Lista de Capacidades*. Una *capacidad* es una *llave* a un objeto para un modo de acceso determinado. Cada sujeto tiene una lista de capacidades asociada que enumera los permisos sobre todos los archivos que puede acceder. Es difícil implementar eficientemente esta técnica donde hay muchos archivos (u objetos) y relativamente pocos usuarios. Hay varias razones para esto, un ejemplo es que hay que actualizar todas las listas cada vez que se crea un objeto, lo cual es muy común en los sistemas más modernos.

Dueño-Grupo-Resto · Una tercera técnica es la que utiliza Unix, *Dueño-Grupo-Resto*. Esta técnica es efectiva cuando podemos prescindir de una gestión fina de los permiso de acceso, ya que sólo permite dos niveles extras además de los permisos para el dueño en cada objeto: *grupo* y *resto*. De esta manera, se especifica para cada archivo los permisos de acceso del dueño, los del grupo en el que está el dueño, y los del resto de los sujetos del sistema. Si se quiere especificar, por ejemplo, que solo dos sujetos puedan acceder a un objeto determinado, entonces no habría otra forma que asignar un grupo que los contenga a ambos.

Listas de Control de Acceso · Esta última técnica que mencionaremos es una de las más efectivas. A cada objeto se le asigna una lista de pares (*sujeto, modo de acceso*), cada uno indicando que al sujeto se le permite acceder al objeto en el modo de acceso indicado. La ventaja principal de este sistema radica en el aspecto implementativo, ya que toda la información para el control de acceso a un objeto determinado se localiza en el mismo lugar, y por lo tanto las operaciones sobre éste involucran elementos todos ubicados en la misma estructura, o al menos en una directamente relacionada con el objeto. Si bien tiene desventajas duales respecto de la técnica anterior, debido a la relación típica de muchos objetos y pocos sujetos en un sistema de cómputo, éstas no constituyen un impedimento tan importante.

Nuestra elección de implementar Listas de Control de Acceso como mecanismo discrecionario recae en su mayor versatilidad para definir políticas de seguridad y en la posibilidad de implementarlas eficientemente.

2.3 Control de Acceso Obligatorio

Mientras que dentro de la categoría DAC podemos encontrar un buen número de alternativas que fueron evolucionando con los años, del lado de MAC una modalidad se ha impuesto como prácticamente la única técnica utilizada para este tipo de control de acceso: Seguridad Multi-Nivel (*MLS, Multi-Level Security*). Tanto que aún es difícil diferenciar entre ambos conceptos debido a la falta de diversidad de técnicas MAC [5].

El *control de acceso obligatorio* es la implementación de una política de seguridad obligatoria. Una política de seguridad obligatoria es aquella en la que la definición de la política y la asignación de atributos de seguridad están estrictamente controladas por los administradores. En otras palabras, si un sistema implementa un mecanismo MAC, parte (al menos) de la información de control de acceso asociada a recursos o procesos puede ser alterada únicamente por una cantidad muy restringida de usuarios (administradores con ciertos privilegios).

Este tipo de control de acceso tiene su origen en el Departamento de Defensa de los Estados Unidos de Norteamérica, donde se sigue usando para la gestión de documentos. En este sistema toda la información se marca mediante una etiqueta que indica su clase de seguridad, al igual que todo individuo. Para que un individuo pueda leer cierta información, éste debe poseer una clase de seguridad que *domine* a la clase de seguridad de la información.

Se dice que una clase de seguridad *domina* a otra si la primera está antes en el orden impuesto para el conjunto de clases de seguridad. El conjunto de clases de seguridad está, en sentido matemático, parcialmente ordenado. Esto quiere decir que dos clases de seguridad pueden no ser comparables, en cuyo caso las respectivas entidades no pueden interactuar. Cualquier orden parcial es válido para usarse en el conjunto de clases de seguridad. El primer orden para el conjunto de clases de acceso utilizado corresponde a aquel definido para el Departamento de Defensa, que se explicará en la siguiente sección.

2.3.1 Política de seguridad del Departamento de Defensa

El Departamento de Defensa de los Estados Unidos de Norteamérica (DoD), tiene una política muy estricta para la administración y almacenamiento manual de información clasificada, que nosotros llamaremos *política de seguridad del DoD* [5]. Esta política fue diseñada primordialmente para proteger la confidencialidad de cierta información militar. Como lo hemos mencionado al referirnos a las políticas de seguridad obligatorias, en esta política cada porción de información (usualmente en forma de documentos) y cada individuo posee una *clase de acceso*. Para determinar si una persona puede tener acceso a un documento, se comparan ambas clases de acceso.

Etiquetas de seguridad

Las clases de acceso se componen de dos partes:

Nivel de seguridad · también llamado *nivel de sensitividad* o simplemente *nivel*, consistente en uno de varios nombres tales como SECRET, TOP SECRET, CONFIDENTIAL, UNCLASSIFIED.

Conjunto de categorías · también llamados *compartimientos*, consiste de uno o más nombres tales como NATO, NUCLEAR, CIA, etc.

El conjunto de niveles de seguridad está linealmente ordenado, en este caso,

UNCLASSIFIED < CONFIDENTIAL < SECRET < TOP SECRET

mientras que las categorías son independientes unas de otras y no están ordenadas.

De esta manera el acceso a cierta porción de información estará legalmente permitido a un individuo si y sólo si su clase de acceso *domina* a la clase de acceso de la información. La relación *domina* se definirá a continuación. La clase de acceso de cada porción de información y de cada individuo sólo puede ser alterada por oficiales especialmente designados para esta tarea.

El propósito de esta política es prevenir el compromiso producido por un individuo capaz de leer información para la que no tiene autorización. En particular, la política no dice nada en cuanto a modificar o destruir esa misma información.

Orden entre etiquetas

Hemos mencionado, como característica de las políticas obligatorias, que existe un orden parcial en el conjunto de clases de acceso. Definiremos ahora la relación *domina*, que tiene lugar entre las clases de acceso del sistema.

Dadas las clases de seguridad como se las definió antes, si n_1 y n_2 son niveles elegidos del conjunto de posibles niveles de seguridad del sistema; y C_1 y C_2 son conjuntos de categorías incluidos en el conjunto de todas las categorías, la clase de acceso (n_1, C_1) *domina* a la clase de acceso (n_2, C_2) si y sólo si $n_1 \geq n_2$ y $C_1 \supseteq C_2$.

De la definición anterior se deduce que en virtud del orden parcial definido por $\subseteq$, las clases de acceso están *parcialmente ordenadas*. En otras palabras, dadas dos clases de acceso, (n_1, C_1) y (n_2, C_2), no siempre será posible decir que o bien (n_1, C_1) *domina a* $n_2, C_2)$ o bien (n_2, C_2) *domina a* (n_1, C_1).

De esta definición surge que individuos e información están clasificados según un criterio que involucra dos dimensiones:

Jerárquica · Está representada por el nivel de seguridad. Dado que se requiere que el conjunto de niveles esté linealmente ordenado, se establece en esta dimensión una jerarquía de documentos e individuos. Los documentos clasificados en los niveles superiores de la jerarquía contendrían la información más crítica, secreta o valiosa para la organización. A medida que se desciende por la jerarquía la importancia o confidencialidad que se le asigna a la información decrece y por lo tanto puede ser vista por mayor cantidad de individuos. Simétricamente, los individuos a los que se les ha asignado un nivel de seguridad alto, se les ha conferido una *altura* tal que les permite ver gran parte de la jerarquía de documentos; cuanto más baja es la altura de un individuo menos le está permitido ver.

No jerárquica · Está representada por el conjunto de categorías. Aunque un individuo puede tener un nivel de seguridad muy alto (o una gran altura) no siempre *necesita*, en su trabajo cotidiano, ver toda la información que tal altura le permite. Por ejemplo, si un comandante (gran altura) está asignado a áreas de la OTAN no debería tener la necesidad de leer información sobre la CIA. Por lo tanto, se combina con la dimensión jerárquica una no jerárquica que hace referencia a las áreas de interés de cada individuo o a las áreas temáticas que toca cada documento.

De esta forma, se permite que cada individuo pueda leer toda la información que su altura le permita pero que sólo es de interés para su trabajo.

En otro orden, cualquier organización que pueda clasificar su información y usuarios según una dimensión jerarárquica y otra no jerárquica con la intención de reglamentar el acceso a aquella puede bien utilizar una variación de la política del DoD.

Una variación trivial, muy simple y evidente, es cambiar los nombres de niveles y categorías por otros que tengan significado en la organización a la que se aplica.

2.3.2 Modelos de Seguridad Multi-Nivel

MLS es acrónimo del inglés *Multi-Level Security* (Seguridad Multi-Nivel). Los modelos MLS son un subconjunto de los modelos MAC, aunque, como hemos dicho antes, no ha habido muchos ejemplos de modelos MAC que no sean modelos MLS. Todos estos modelos de seguridad provienen de las investigaciones para el DoD en las décadas de los '70 y '80, de ahí su orientación a la confidencialidad.

Los modelos MLS han sido desarrollados como descripciones matemáticas y/o lógicas de adaptaciones a los sistemas de cómputo de la política de seguridad del DoD. El modelo MLS más conocido, estudiado e implementado es el descrito

por D. Elliott Bell y Leonard J. LaPadula en 1972 [9, 8] como parte de las investigaciones que desarrolló The Mitre Corp. para el DoD.

Las medidas base de este tipo de esquemas son las siguientes:

- Ciertos atributos de seguridad de procesos y archivos sólo pueden ser alterados por administradores y no por los usuarios ordinarios (MAC).
- Se define sobre el conjunto de archivos y procesos un orden parcial.
- El sistema sólo permite que la información fluya en sentido ascendente según el orden mencionado.
- El orden parcial se define, precisamente, utilizando aquellos atributos de seguridad que sólo pueden ser modificados por los administradores.

Un modelo MLS típicamente incluye una descripción abstracta del sistema de cómputo que se programará y una descripción de las propiedades de seguridad que se espera que el sistema posea.

En el modelo abstracto del sistema de cómputo se destacan las siguientes entidades típicas:

Recursos · Un conjunto de recursos de cómputo a proteger

Sujetos · Un conjunto de sujetos capaces de acceder a dichos recursos.

Modos · Un conjunto de modos de acceso (lectura, escritura, etc.).

Clase de acceso de recursos · Función que asocia a cada recurso su clase.

Clase de acceso de sujetos · Función que asocia a cada sujeto su clase.

Operaciones · Operaciones para acceder a los recursos protegidos.

Dentro de las propiedades deseadas del modelo, usualmente se encuentran:

MAC · Un predicado que establece que las clases de acceso de sujetos y recursos sólo pueden ser alterados por administradores y en condiciones especiales.

Seguridad simple · Un predicado que debe ser verificado por todas las operaciones que establece que en todo momento cualquier sujeto que está accediendo a un recurso lo hace sólo porque la clase de acceso del sujeto domina a la del objeto. Esta propiedad es de carácter más bien estático ya que sólo involucra la identidad del sujeto y del objeto pertinentes.

Confinamiento · Si existen operaciones que escapan al control del sistema de seguridad, como sucede en la mayoría de los casos, se debe agregar otro predicado denominado *confinamiento*, que se decribirá a continuación. Esta propiedad, contrariamente a la de seguridad simple, es de carácter más dinámico, ya que involucra un análisis de los objetos accedidos por cierto sujeto al momento del requerimiento.

Debido a que no es posible controlar toda operación de movimiento de datos dentro de la memoria, el primer modelo MLS definió la propiedad ahora llamada *confinamiento*. BLP introdujo la entonces llamada propiedad-★ para lidiar con ciertos casos en los cuales el flujo de información quedaba fuera del alcance del control del modelo.

Así, la propiedad de confinamiento sigue:

> *Para todo sujeto, la clase de seguridad de los archivos a los que tiene permiso de escritura debe superar la clase de cualquier archivo al que tenga permiso de lectura*

La intención de esta regla es evitar que una vez abierto cierto archivo con una clase de acceso alta se abran otros archivos con clase de acceso inferior y se copie el contenido del archivo de clase alta en los de clase baja. Subyacen a esta idea dos hechos:

- Se han considerado dentro del conjunto de recursos a proteger sólo algunos recursos del sistema y no todos, por ejemplo, la memoria usada por los procesos no está gestionada como recurso protegido.
- El control de acceso se realiza al momento de pedir acceso a cierto objeto.

En general esto es así debido a que la implementación de controles semejantes sobre la memoria (dado que puede haber flujo de información a través de ella) no es posible o es muy complicado en la práctica.

Del enunciado de esta regla se deduce que no se puede determinar si un proceso pasará el control de confinamiento al intentar abrir un archivo con sólo analizar la relación entre la clase de acceso del proceso y la del archivo. Precisamente, la clase de acceso del proceso no juega, en esta regla, ningún papel.

La mayor parte de los modelos MLS han sido especificados y verificados utilizando métodos formales ya que se supone que luego serán implementados en sistemas destinados a organizaciones para las cuales la protección de la información es una actividad de gran valor y, en ocasiones, crítica.

2.4 Caballos de Troya

Los caballos de Troya[1] [5, 1] no fueron descubiertos como ruta para penetrar la seguridad de un sistema sino hasta bastante tarde en la historia de los sistemas de cómputo. Esta forma de ataque es conocida y estudiada desde mediados de la década del '60. Como su contraparte histórica, refiere a una técnica para atacar sistemas de cómputo desde adentro en lugar de hacerlo frontalmente contra barreras defensivas poderosas. Aun así, los caballos de Troya atacan sistemas sin necesidad de que éstos posean fallas en sus sistemas de protección. Un caballo de Troya es un programa cuya ejecución resulta en efectos secundarios indeseados no anticipables por el usuario. Normalmente se presenta como un programa útil pero contiene funciones que atacan al sistema; estas funciones, obviamente, no están documentadas ni publicitadas. La utilidad manifiesta del caballo de Troya se basa en su capacidad para hacer que usuarios inocentes ejecuten el programa; en ese mismo acto activando, sin intención, ni siquiera conocimiento, las funciones de ataque.

Resaltaremos algunos puntos importantes:

- Ni los administradores ni los usuarios de la organización atacada son conscientes de la existencia de las funciones de ataque presentes en el programa.
- Ni los administradores ni los usuarios de la organización tienen intenciones de violar la política de seguridad revelando información a terceros. A lo sumo algún usuario o administrador que no tiene acceso a cierta información es quien se las arregla para instalar el caballo de Troya en el sistema, pero quienes lo usan (y tienen acceso a la información importante) no saben que están usando un programa de ataque.
- Cualquier software que se ejecute a nivel de usuario puede ser un caballo de Troya: editores de texto, planillas de cálculo, comandos del sistema operativos, programas de administración del sistema, aplicaciones corporativas, servidores o clientes TCP/IP, juegos, software de desarrollo, etc.
- No es necesario que exista una falla en el sistema de protección (en todo caso el sistema está *fundamentalmente* fallado o no ha sido preparado para soportar este tipo de ataques).

Digamos que el caballo de Troya es ejecutado por cierto usuario que tiene acceso a la información que interesa al atacante. Al hacerlo, el programa se convierte en un proceso a nombre del usuario como cualquier programa normal. Por lo tanto, dicho proceso tiene acceso a todos los recursos que cualquier otro proceso del usuario posee. En particular puede acceder a los archivos, tablas, bases

[1] El término *Caballo de Troya* fue utilizado por primera vez por Dan Edwards.

de datos, etc. que contienen la información a obtener. Mientras el programa presta la función útil por la que el usuario lo ejecutó, el caballo de Troya activa sus funciones de ataque, efectúando una o varias de las siguientes acciones dependiendo de la "distancia" que separe al atacante de la víctima y de qué tan profundo sea el daño que se desee realizar:

- Copia la información valiosa en archivos, tablas, etc. a nombre del atacante en el mismo sistema.

- Envía la información por mail o cualquier otro protocolo capaz de sortear el o los Firewalls interpuestos entre la organización e Internet.

- Cambia la información de control de acceso de los archivos, tablas, etc. donde se almacena la información secreta de manera tal que más tarde el atacante pueda acceder a ella normalmente.

- Modifica el código objeto de cualquier programa escrito por el usuario de manera tal que estos otros programas se convierten en nuevos caballos de Troya (es decir, aquí se utilizan técnicas propias de los virus informáticos pero con una diferencia muy importante: al no ser programas de ataque indiscriminado, sino todo lo contrario, es muy poco probable que alguien lo detecte y lo reporte como un virus de manera que sean reconocidos por programas anti-virus).

De los párrafos precedentes debe resultar claro que un caballo de Troya no requiere una falla en el sistema para lograr su cometido; y que como arma de ataque, es eficaz y poco detectable, especialmente en los sistemas de uso masivo.

2.5 DAC, MAC y Caballos de Troya

Las razones por las que se incluyen o implementan ambos modelos, uno DAC y otro MAC, en el sistema, pueden ser diferentes en cada caso; en el nuestro:

DAC · Brindar mayor libertad a los usuarios para proteger y compartir información cuando la política obligatoria no restringe estas acciones en favor de una causa mayor.

MLS · Permitir expresar políticas multi-nivel. Resistir ataques contra la confidencialidad basados en caballos de Troya.

Integración de los modelos

La manera de integrar ambas políticas es arbitraria según la política de seguridad que se desee tener. Asi, las siguientes condiciones delinean una elección común, la mera conjunción de las políticas:

- El proceso tiene los permisos DAC necesarios para ejecutar acceder al recurso.
- El proceso debe verificar seguridad simple con respecto al recurso.
- El proceso debe verificar confinamiento con respecto al recurso que se intenta acceder y en relación con todos los otros recursos que aún tiene abiertos.
- Sólo administradores MAC pueden cambiar la clase de los objetos.

Naturalmente, un sistema con estas caracterísiticas es más restrictivo que un sistema que impone sólo uno de los modelos; el balance se suele inclinar hacia la usabilidad según se pierda protección y viceversa.

DAC y MAC versus Caballos de Troya

Las técnicas DAC son esencialmente vulnerables a los ataques mediante caballos de Troya, y aun con MAC es muy difícil evitarlos, incluso controlarlos [5]. Esto se debe a que el caballo de Troya, al heredar todos los permisos de acceso del sujeto, obtiene la posibilidad inmediata de acceder a todos los objetos a los que éste tiene acceso. Si a un usuario se le permite modificar los *derechos de acceso* de *sus* objetos, entonces también puede hacerlo un caballo de Troya, y el ataque estaría pasando desapercibido.

La única manera de restringir las operaciones de un caballo de Troya es restringiendo las operaciones de los usuarios mismos sobre los objetos. La administración de dicha política debe estar fuera del alcance de los usuarios porque de otra manera podría ser alterada también por un caballo de Troya, de ahí las características de un sistema MAC, y las razones por las que éstos son efectivos para resistir los ataques de este tipo.

Por eso es especialmente dificil controlarlos en sistemas de propósito general, ya que resulta dificultoso establecer políticas obligatorias con buena usabilidad y a la vez efectivas.

Un sistema MLS impide que un caballo de Troya haga *descender* información, sólo puede hacerla *ascender* (o a lo sumo dejarla al mismo nivel). Por esto ningún movimiento de datos puede hacer que alguien que no tiene acceso a cierta información pueda tenerlo en virtud de las actividades del caballo de Troya.

Notar que el punto debil más importante donde un sistema MAC pierde la utilidad de manera absoluta ante un caballo de Troya es si este logra ingresar y ejecutarse en el estado especial del sistema desde el cual se administra la política de seguridad. De ser así, la capacidad del sistema MAC para resistir el el ataque equivale a la de cualquier sistema DAC.

Por esto debe tenerse especial cuidado en la definición de los roles y las responsabilidades para la administración de las etiquetas de seguridad. Una mala administración de este punto puede llevar al estado antes descrito dejando al sistema vulnerable.

Capítulo 3

Bell-LaPadula

El modelo Bell-LaPadula fue el primer modelo formal para seguridad, diseñado para administrar la seguridad de la información en el Departamento de Defensa de los Estados Unidos de Norteamérica.

En este capítulo explicaremos las bases de este modelo en forma abreviada. No enunciaremos los teoremas que aparecen en los artículos ni tampoco sus pruebas. Lo explicaremos en una forma moderadamente detallada con el objetivo de que sirva de base conceptual para la especificación principal de este documento, cuyo núcleo y centro de atención es este modelo.

3.1 Contenido de los artículos

Presentado en 1973 por Leonard J. LaPadula y D. Elliot Bell, el modelo implementa una política de seguridad multi-nivel que, al momento de su aparición, reflejó en los sistemas informáticos los mecanismos de control de acceso ya existentes en el departamento. Las ideas principales del modelo se publicaron en dos artículos [9, 8], el primero definiendo el comportamiento más abstracto del modelo, configurando un marco de trabajo para el desarrollo del segundo; y este último bastante más refinado y especializado a los requerimientos de un sistema de cómputo estándar. Es en éste en donde se presenta un modelo suficientemente práctico como para ser implementado.

El primer artículo introduce un modelo matemático para sistemas de cómputo seguros basado en la teoría general de sistemas. Se introduce aquí una noción abstracta de sistema seguro y se prueban varias propiedades del modelo diseñado.

Luego, en el segundo artículo, se presenta un conjunto de reglas de operación (funciones que publica el sistema a sus usuarios) que garantizan que el sistema siempre quede en un *estado seguro*. Estas reglas se construyen sobre un modelo algo refinado respecto de su antecesor. Así, a todas es fácil transformarlas en algoritmos implementables en cualquier sistema de cómputo digital.

Es en el último artículo en donde se define el modelo en el que se basa este trabajo.

3.2 Modelo de sistema

Describiremos ahora las características del modelo de sistema que utiliza Bell-LaPadula (*BLP*). Esta sección sirve de justificación para las formalizaciones que se encuentran en las siguientes secciones y también para las de nuestra especificación, que abordaremos en el siguiente capítulo.

Una de las diferencias entre el primer y segundo trabajo de Bell y LaPadula es que en el segundo [8] se trabaja sobre un modelo de sistema de cómputo en el cual ya se definen un conjunto determinado de *modos* en los cuales los sujetos pueden acceder a los objetos.

3.2.1 Modos de acceso

Se considera un sistema de cómputo con cuatro tipos básicos de acceso a documentos:

Sólo lectura · Se le permite al sujeto acceder a un objeto para leer su contenido.

Por ejemplo, cuando el objeto es un archivo.

Sólo escritura · (o *adjunción*) Se le permite al sujeto acceder a un objeto para escribirlo. No puede acceder a su contenido original, tampoco puede cambiarlo ni borrarlo, y como no puede leer su tamaño ni tampoco ubicarse dentro del mismo en una posición particular, sólo agrega datos al final.

Lectura/escritura · Se le permite al sujeto acceder a un objeto tanto para leerlo como escribirlo.

Ejecución · Se le permite al sujeto ejecutar un objeto; que es típicamente un programa, o podría ser, por ejemplo, un directorio, u otro tipo de objeto que permita algún tipo de activación.

Adicionalmente, se considerará la existencia de un quinto modo de acceso, a través del cual se pueden modificar las propiedades relacionadas con la identidad del objeto, pero no así su contenido.

Control · Un sujeto puede alterar los atributos de seguridad de un objeto; casi siempre sólo aquellas relevantes al sistema operativo en uso, y muchas veces administradas sólo por este último. A través de este se podría, por ejemplo, alterar la política de seguridad aplicada al mismo, pero no sus datos internos (en caso de tenerlos).

La política de seguridad se basará en permitir o restringir accesos de estos tipos a los objetos. De esta forma, la lista anterior define la granularidad de la política en lo que respecta a la diversidad de formas en las cuales se puede interactuar con un objeto.

3.2.2 Requerimientos y Decisiones

Utilizaremos el término *requerimiento* para referirnos al pedido que un usuario hace al sistema. Asumiendo que todo proceso es delegado de algún usuario, estaremos siempre hablando de procesos que realizan requerimientos al sistema para obtener acceso a los objetos, no pudiendo los usuarios interactuar con el sistema sinó a través de éstos.

Se tienen inicialmente cuatro tipos de requerimientos:

Obtener/liberar acceso a un objeto · Este tipo de requerimiento altera la parte del estado que indica qué sujeto esta potencialmente accediendo a qué objeto.

Entregar/quitar permiso para pedir acceso · Permiten modificar los derechos de acceso.

Crear un objeto · Mediante este tipo de requerimiento se crean objetos.

Eliminar un objeto · A través de este requerimiento se eliminan objetos.

Sin embargo, BLP decide tomar, por simplicidad, una perspectiva algo diferente para crear y eliminar objetos. En lugar de ver la primera de estas operaciones como una creación, se modelará este comportamiento como si fuera una *activación*. Se asume que todos los objetos existen y que no es posible utilizarlos hasta que no son *activados* mediante un requerimiento de activación. Al mismo tiempo, existe una función para cambiar las etiquetas de seguridad de objetos no activos. Esto se debe a un aspecto técnico: de esta forma BLP evade elegantemente variar el dominio de las funciones que utiliza para la clasificación y el tamaño de la matriz de acceso.

Análogamente a la creación de objetos, la eliminación se hace mediante un requerimiento de *desactivación*, que elimina automáticamente el registro de todos los derechos de acceso al objeto así como toda otra referencia al mismo.

Aunque BLP ha modelado de esta forma la creación y eliminación de objetos, esta última no se utilizará en nuestra especificación, ya que el aspecto técnico que forzó a los autores de BLP a modelarlo de esta forma no resulta un problema bajo nuestro lenguaje de especificación o nuestro mecanismo de prueba.

Todo requerimiento de un sujeto para acceder a un objeto puede ser prohibido por la política de seguridad, en cuyo caso el requerimiento falla y el sujeto no puede obtener el acceso al objeto como era su intención. Esto introduce el concepto de fallo en un requerimiento y lo modelaremos mediante las *decisiones*.

Hablaremos de *decisiones* para referirnos a la respuesta del sistema luego del requerimiento de un usuario, por ejemplo, para cambiar los permisos de acceso. En este caso, una decisión podría determinar la entrega del modo de acceso requerido, o denegarla.

Luego de que un requerimiento tiene lugar, el sistema responde con una *decisión*. BLP define las siguientes posibles decisiones:

Si · Se permite la ejecución de la operación asociada al requerimiento.

No · No se permite la ejecución de la operación asociada al requerimiento.

? · Indica que el sistema no entiende el requerimiento.

3.3 Propiedades de Seguridad

El modelo BLP basa su política de seguridad en la del DoD antes mencionada. El enunciado de la misma corresponde, naturalmente, a una definición de com-

promiso de seguridad con sentido militar o gubernamental: falta de confidencialidad, es decir, divulgación no autorizada de información. Un compromiso de seguridad no debe tener lugar dentro del sistema de cómputo. De este predicado se deriva la definición de la política de seguridad de BLP.

Una de las desventajas de las perspectivas menos formales para modelar un sistema seguro es que la definición de seguridad suele ser mantenida en lenguaje natural y es interpretada según el caso para implementar los chequeos correspondientes. Así, las políticas suelen ser poco claras y es difícil separarlas de otros conceptos.

Mediante la formalización, este modelo permite tanto una descripción precisa de su funcionamiento como un mecanismo de pruebas para verificar correctas implementaciones en los sistemas anfitriones del mismo.

3.3.1 Clases de Seguridad

BLP requiere que se etiquete toda la información con su clase de seguridad. La estructura de esta etiqueta se hereda de la política del DoD antes definida. Es decir, se tienen en la etiqueta de seguridad un conjunto de categorías y un nivel de seguridad tal como fue descripto para la política de seguridad del DoD.

La idea es etiquetar los *contenedores* de información, por ejemplo, los archivos. Así lo mismo para el resto de los recursos del sistema: impresoras, pantallas, vías de información en general, fuentes de información o destinos. Se etiqueta cada recurso y de las misma forma los sujetos.

3.3.2 Seguridad Simple

La condición de seguridad representa la forma más esencial de la política de seguridad en el modelo. Se da el nombre de *seguridad simple* a la propiedad de cumplir con la condición de seguridad.

La *condición de seguridad* corresponde a:

> *prevenir que todo sujeto de una cierta clase de seguridad tenga acceso de lectura a todo objeto con una clase que la exceda, es decir, no esté dominada por la anterior*

Esta condición es definida en términos inmediatos y no contempla accesos no autorizados *probables* o *potenciales*.

Si esta propiedad se verifica en todo estado del sistema entonces tentremos certeza de que la seguridad no está comprometida en el contexto de la definición dada. Sin embargo, y debido al modelo de cómputo utilizado, esta aserción no

basta para asegurar la información contra toda falla en la protección de la confidencialidad.

Es imposible monitorear todo paso de ejecución de un programa; por lo tanto pueden existir movimientos de memoria u otras operaciones no capturadas por el modelo que atenten contra nuestro objetivo. La siguiente sección introduce una propiedad que también impone BLP que cubre esto último, pero la esencia de la política de seguridad permanece intacta y se refleja en su forma más clara en la condición recién enunciada.

3.3.3 Propiedad-★

En ciertos casos, la interacción de múltiples accesos, ninguno de los cuales representa un compromiso de seguridad, posteriormente puede llevar a uno. El aspecto a destacar de estos casos es la pérdida de control por parte del sistema de seguridad.

Este escenario representa el caso mencionado: un sujeto tiene derecho de lectura de un objeto y abre otro para sólo escritura (no se puede obtener información del objeto) con clasificación menor que la del objeto que ya había abierto anteriormente. En consecuencia, el sujeto podría leer información de un nivel superior y escribirla a un objeto de nivel inferior y por lo tanto sujetos con menores niveles de clasificación podrían leer esta información. BLP soluciona este problema con la denominada propiedad-★:

Para un sujeto dado, todo objeto abierto para sólo escritura supera en clasificación a todos los objetos abiertos para lectura

De esta forma, BLP cubre el espacio en donde pierde el control sobre el movimiento de información, aunque con una restricción bastante fuerte.

3.4 Aplicando BLP

Para definir su política, BLP formaliza las nociones de sistema y estado de sistema para construir los predicados involucrados. Un sistema procesa requerimientos por parte de los usuarios para acceder a los objetos y toma decisiones en base a la política de seguridad para llegar a conclusiones que impactan sobre la respuesta a los requerimientos.

Al aplicar este modelo a un sistema en la vida real debemos modelar de manera similar el estado del sistema que queremos proteger, y luego especificar las funciones disponibles a los usuarios. También debemos especificar los predicados que definen la política de seguridad, así podremos analizar si estas últimas sirven o no para evitar compromisos de seguridad en el sistema resultante.

Bell y LaPadula muestran la línea a seguir especificando un conjunto de *reglas* que representan los requerimientos que un usuario puede efectuar y probando formalmente que dichos servicios no introducen compromisos de seguridad.

3.4.1 Reglas de Operación

BLP define las siguientes reglas de operación para su modelo de sistema:

get-read · obtener acceso en modo lectura.

get-append · obtener acceso en modo escritura.

get-execute · obtener acceso de ejecución.

get-write · obtener acceso en modo lectura-escritura.

release-read/write/all/execute · liberar acceso en modo x.

give-read/write/all/execute · entregar un derecho de acceso.

rescind-read/write/all/execute · rescindir derecho de acceso.

change-f · cambiar la funcion de clasificación entera.

create-object · crear un objeto.

delete-object · eliminar un objeto.

Nuestra especificación se basa en estas reglas dadas por BLP. La mayoría son iguales pero algunas han sido alteradas por conveniencia.

Para BLP una *regla de operación* es una función que se aplica sobre un requerimiento y un estado de sistema y lo transforma retornando otro estado y una decisión. Se ha formalizado de la siguiente manera; tomemos, por ejemplo, *get-read*:

$$
\begin{aligned}
&\textit{get-read} : \rho_1(R_k, v) \equiv \\
&\quad \textbf{if } \theta_1 \neq \emptyset \ \vee \ \gamma \neq g \ \vee \ \underline{x} \neq \underline{r} \ \vee \ \theta_2 = \phi \\
&\quad\quad \textbf{then } \rho_1(R_k, v) = (\underline{?}, v); \\
&\quad \textbf{if } \underline{r} \notin M_{ij} \ \vee \ [f_1(S_i) < f_2(O_j) \ \vee \ f_3(S_i) \not\supseteq f_4(O_j)] \\
&\quad\quad \textbf{then } \rho_1(R_k, v) = (\underline{no}, v); \\
&\quad \textbf{if } U_{\rho_1} = \{O : O \in b(S_i : \underline{w}, \underline{a}) \wedge \\
&\quad\quad\quad [f_2(O_j) > f_2(O) \ \vee \ f_4(O_j) \not\subseteq f_4(O)]\} = \emptyset \\
&\quad\quad \textbf{then } \rho_1(R_k, v) = (\underline{yes}, augb(R_k, v)); \\
&\quad\quad \textbf{else } \rho_1(R_k, v) = (\underline{no}, v); \\
&\quad \textbf{end.}
\end{aligned}
$$

La tupla $v = (b, M, f)$ representa un estado del sistema y $R_k = (\theta_1, \gamma, \theta_2, O_j, \underline{x})$ es una tupla con toda la información asociada al requerimiento hecho por el sujeto. Las variables que se nombran sueltas en la especificación de *get-read* son las asociadas a los vectores R_k o v. Las funciones f_1, f_2, f_3 y f_4 son las funciones de clasificación del sistema:

f_1 · da para cada sujeto su nivel de seguridad asociado

f_2 · da para cada objeto su nivel de seguridad asociado

f_3 · da para cada sujeto su conjunto de categorías asociado

f_4 · da para cada objeto su conjunto de categorías asociado

Estas funciones son la forma de evaluar las *etiquetas de seguridad* del sistema en el contexto de la formalización que usaron Bell y LaPadula.

El análisis de las operaciones se ve algo complicado bajo esta formalización por pretender ser R_k un vector que representa *cualquier* operación (y sus parámetros) del sistema. En consecuencia, hay operaciones para las cuales algunos parámetros no son utilizados en absoluto y contaminan la claridad de la especificación. A su vez los parámetros poseen nombres algo genéricos dada su utilización de semántica múltiple y esto genera aun menos autoexplicación en la espeficicación de la mayoría de las operaciones.

De cualquier modo, esta definición se entiende de la siguiente manera: Primero se espera que el parámetro θ_1 sea ϕ (sujeto nulo —ya que este parámetro no es usado), que γ sea g (el símbolo que representa el requerimiento de obtener o dar un modo de acceso a algún sujeto), que $\underline{x}$ sea $\underline{r}$ (el símbolo que representa el modo de acceso *read*) y que θ_2 (el sujeto) no sea ϕ; de otra forma la operación no se ejecuta y se retorna $\underline{?}$ como decisión final; lo cual significa que los parámetros no corresponden a esta operación y por ende no se ejecutó. Esto último es cubierto por la primera estructura **if**.

Segundo se chequea que el sujeto indicado por S_i (que se asume indicado a través del parámetro θ_2), esté en condiciones de obtener el modo de acceso *read* según lo determina la propiedad de *seguridad simple*. Es decir, que el nivel del sujeto sea mayor o igual al nivel del objeto al que desea acceder ($f_1(S_i) \geq f_2(O_j)$), y que el conjunto de categorías del objeto este contenido por el conjunto de categorías del sujeto ($f_3(S_i) \supseteq f_4(O_j)$).

Se debe notar que además de las dos condiciones antes mencionadas (relacionadas con la propiedad de seguridad simple) aparece un predicado extra: $\underline{r} \in M_{ij}$. Este predicado está asociado a un mecanismo de control de acceso discrecionario que utilizaron los autores de BLP, una *matriz de acceso*. Debido a que en nuestra especificación hemos reemplazado la matriz de acceso por el mecanismo

de Listas de Control de Acceso, este predicado diferirá levemente en nuestra versión de la operación. Dejaremos ahora este detalle para más adelante.

Si el sujeto y el objeto asociados no validan la propiedad de seguridad simple, entonces la respuesta del sistema es $\underline{no}$. Caso contrario se valida el cumplimiento de la última condición necesaria: la propiedad-★.

BLP usa aquí una sintaxis especial en el término $b(S_i : \underline{w}, \underline{a})$. Este término se valua al conjunto de objetos a los que el sujeto S_i está accediendo en modo *write* o *append*. Si todos estos objetos poseen una clase de seguridad que supera la del objeto O_j, entonces la propiedad-★ quedará validada luego de la ejecución de la operación si valía antes de su ejecución.

Por último, si todas las condiciones son validadas, la operación da como resultado $(\underline{yes}, augb(R_k, v))$. La función *augb* entrega el modo de acceso *read* al sujeto asociado sobre el objeto requerido.

3.4.2 Verificación de las Propiedades de Seguridad

Bajo la asunción de que un sistema sólo cambia de estado a través de sus ya enunciadas reglas de operación, sólo estas últimas pueden introducir un compromiso de seguridad.

Así, BLP certifica que un sistema es seguro probando que ninguna de las reglas puede introducir un compromiso de seguridad. Una vez que se asume o se prueba que el estado inicial de un sistema es seguro, sólo hace falta probar que si una regla se aplica sobre un estado seguro entonces da como resultado otro estado seguro.

Las pruebas en BLP están divididas en dos partes, la preservación de la seguridad simple y la preservación de la propiedad-★. Las pruebas finales de estos predicados para cada una de las operaciones no son complejas porque se presentan en el artículo un conjunto de teoremas que ayudan a reducir el tamaño y la complejidad de las mismas. No las presentaremos aquí porque su entendimiento implicaría conocer también estos teoremas y sería demasiado extenso para el objetivo de este capítulo.

Un trabajo análogo al que Bell y LaPadula realizaron con la especificación del modelo y las pruebas recién mencionadas se llevará a cabo en los dos capítulos siguientes, incluyendo la prueba de preservación de seguridad simple y propiedad-★ para una de las operaciones desarrolladas. Las pruebas se harán formalmente, no como en el artículo de BLP, en las que se utiliza demasiado lenguaje natural y tienen una forma bastante simplista. Nosotros, en cambio, las enunciaremos y probaremos mediante mecanismos de prueba puramente sintácticos, a través del asistente de pruebas de Z/EVES.

3.5 Conclusión

Debido a que es un modelo largamente estudiado, elegiremos BLP para implementar MLS en nuestro módulo de control de acceso.

Capítulo 4

DAC + MLS en un Servidor de Aplicaciones

Este capítulo presenta el contexto para el que se quiere concebir el módulo de control de acceso y define la funcionalidad a requerirse de este último.

El objetivo principal de este trabajo es especificar un módulo de control de acceso y verificar sus propiedades de seguridad. En favor de este fin se ha elegido fijar un contexto anfitrión típico para nuestro futuro resultado. Entre otras cosas, la elección prematura de una arquitectura en la cual se pueda utilizar el módulo nos sirve para derivar detalles menores que podemos esperar de su funcionalidad. El contexto elegido es un servidor de aplicaciones, dado que representa un escenario concreto actual y muy usado. Comenzaremos este capítulo introduciendo el concepto de servidor de aplicaciones y luego encararemos la definición de los requerimientos del módulo de control de acceso. En el siguiente capítulo tomaremos entonces estos últimos para transformarlos en una especificación verificable formalmente.

4.1 Servidores de Aplicaciones

La idea original de Tim Berners-Lee al inventar parte de la tecnología y construir la infraestructura para lo que luego se convertiría en la World Wide Web distaba en magnitudes sorprendentes de la concepción de esta última siquiera como una posibilidad.

La WWW fue construida sobre una familia de protocolos no propietarios simples y robustos. El protocolo HTTP (HyperText Transfer Protocol), las URL (Universal Resource Locator) y el lenguaje HTML (HyperText Markup Language). Las características de esta combinación hicieron de la WWW el catalizador ideal para un conjunto de líneas que habían estado circulando en el ámbito corporativo ya desde algunos años. La familia de herramientas que ahora llamamos *Servidores de Aplicaciones* es el resultado de dos lineas de desarollo originalmente independientes que convergieron en la WWW [14]. Una de estas lineas comienza con las herramientas de desarrollo de sitios web, y la otra creció desde las arquitecturas de aplicaciones de multiples estratos.

Las *arquitecturas de tres estratos*, representando un ejemplo de la evolución natural del paradigma cliente-servidor, consisten en un o más sistemas del último estrato (p.ej. servidor de BBDD), una interfaz de usuario, y un estrato intermedio que ejecuta la lógica del negocio. El estrato intermedio es responsable de integrar los datos provistos por los sistemas del último estrato en el sistema completo. Maneja seguridad, provee una interfaz de programación a clientes, y otras responsabilidades más genéricas que fueron desapareciendo con la evolución de estos sistemas, que ahora son delegadas a una plataforma estándar.

Un *servidor de aplicaciones* es el estrato intermedio de una arquitectura de tres estratos: se comunica con los sistemas del último estrato, con los clientes (del primer estrato) (casi siempre, pero no necesariamente, clientes web), y provee un marco de trabajo sobre el cual la lógica del negocio se pueda desarrollar. El mismo se puede sincronizar y combinar con el servidor web para procesar los requerimientos que hacen los clientes web. Cuando un cliente hace un requerimiento que llega al servidor web, este se encarga de enviar la información necesaria al servidor de aplicaciones, el cual realiza las acciones necesarias y luego envía la respuesta de vuelta al servidor web. Por último, este envía la información procesada de vuelta al cliente [6]. El resultado es una arquitectura modular, altamente escalable, y robusta.

4.1.1 Seguridad en un Servidor de Aplicaciones

Además de abrir nuevas vías de acceso a los usuarios, Internet está cambiando la arquitectura de los sistemas de cómputo actuales. Lejos de estar restringido por

el estilo monolítico de los mainframes, el software moderno tiende a aprovechar vastamente la potencialidad del nuevo medio: el procesamiento distribuido. Pero el tremendo nuevo potencial que conlleva este nuevo paradigma también introduce nuevas necesidades en el ámbito de la implementación de una política de seguridad. El acceso a datos a nivel global que ofrece Internet es también una fuente de preocupación: para poder usar Internet, tecnología cuya razón de ser es abrir continuamente nuevas vistas, de forma segura, es necesario encontrar continuamente nuevas formas de implementar controles de seguridad.

No es posible en general agregar seguridad a la manera de una enmienda (*patch*). Buenas prácticas de seguridad incluyen un número considerable de atributos deseables, siendo importante para todos ellos un análisis para el caso particular [16].

Propiedades de seguridad deseadas

Destacaremos los siguientes mecanismos, uno de los cuales provee el módulo de control de acceso:

Autenticación · Una tarea esencial de un sistema de seguridad es autenticar los sujetos, que en el caso de un servidor de aplicaciones o cualquier arquitectura distribuida, puede tener lugar en forma remota. Debe darse particular énfasis en contextos en donde pueden ocurrir ataques mediante caballos de Troya.

Autorización · Despues de la autenticación, la siguiente característica deseable es un servicio de autorización y una forma de implementar control de acceso. Los administradores deben poder decidir qué nivel de acceso se aplica a un sujeto dado. Relacionado con esto, otro atributo también importante es la granularidad ofrecida en la definición del área accesible por cada sujeto.

Auditoría · Un buen sistema de seguridad debe también proveer una forma de implementar auditoría. En el caso de que una incursión tenga lugar, el registro de auditoría debe poder asistir a los administradores en sus esfuerzos por resolver el problema.

Aparte de los atributos antes mencionados, dado que ningún sistema es totalmente impenetrable, mencionaremos tres tópicos generales acerca de la forma de lidiar con ataques al sistema en última instancia. El objetivo de una estrategia de seguridad sólida es atacar en forma balanceada los siguientes puntos:

· Minimizar los costos potenciales de una intrusión.

· Minimizar el impacto del ataque en los usuarios del sistema.

- Maximizar la dificultad de efectuar un ataque.

El esfuerzo empeñado en las tres anteriores depende de las necesidades de cada sistema y de los servicios que presta.

Usabilidad versus Seguridad

Una característica típica de los sistemas de cómputo es que su *usabilidad* corresponde inversamente a su *seguridad*. Medidas muy restrictivas sin duda reducen la productividad de los usuarios del sistema. En casos extremos pueden buscar maneras de evitarlas, por ejemplo cuando se encuentran bajo mucha presión, exigidos de tener resultados inmediatos. Sabiendo que una de las trabas a la agilidad son los controles de seguridad, seguramente querrán evitarlos. Y si no pueden hacerlo, entonces tal vez busquen la cooperación del administrador que les impone la seguridad.

Diferentes modelos ofrecen diferentes restricciones y niveles de usabilidad. Una característica tecnológica, externa al modelo, que interfiere con el balance entre los dos anteriores es la plataforma de implementación. Esto alude nuevamente al problema del código ejecutable nativo mencionado en la sección anterior. Las plataformas que ejecutan código nativo dificultan en mayor medida la implementación de, a la vez, restricciones efectivas y alta usabilidad, debido a la pérdida de control por parte del modelo. Plataformas que ejecutan código interpretado poseen una mayor versatilidad por tener mayor control del código en ejecución.

La propiedad de confinamiendo antes mencionada para los modelos MLS es un ejemplo de una medida muy restrictiva para ofrecer alta seguridad en sistemas que ejecutan código nativo.

4.1.2 Arquitecturas de Control de Acceso

Seguridad se ha implementado en sistemas de cómputo en una variedad de formas, lugares, y niveles de abstracción con diferente énfasis. Por ejemplo, a nivel de sistema operativo, de base de datos, de red, de *middleware*, etc. Estudios más recientes acompañan la evolución del software en su crecimiento hacia los entornos distribuidos y analizan el problema de implementar seguridad en estos contextos [4].

La estructura tradicional para implementar control de acceso puede verse usando el concepto de monitor de referencias. Un monitor de referencias es la entidad de software a cargo del control de acceso, en otras palabras, es el responsable de mediar los accesos por parte de los sujetos a los objetos.

Asumiendo que el monitor de referencias es la única interfaz a los objetos, éste es el responsable de determinar qué ocurre con cierto requerimiento de acceso. La decisión, en el caso de un sistema de seguridad, se delega en la política definida para el sistema. La política de seguridad, a veces vista como un conjunto de reglas que hablan acerca de los permisos de acceso, se obtiene de algún tipo de repositorio que el monitor de referencias consulta.

El monitor de referencias evalua la información dada y toma una decisión, permitiendo o denegando el acceso. En muchos modelos la información disponible para la decisión se concentra en tres grupos principales, la indentidad del sujeto, la identidad del objeto, y el tipo de acceso requerido.

Las tres características fundamentales de un monitor de referencias, originalmente identificadas por J. P. Anderson, son [1]:

Completitud · Ser invocado para toda referencia de un sujeto a un objeto

Aislamiento · Estar protegido contra modificaciones ilegales.

Verificabilidad · Ser pequeño, bien estructurado, simple, y entendible.

Es necesario que el un módulo de control de acceso represente un monitor de referencias entre los sujetos y los objetos. Si quedan a disposición del usuario solo operaciones que mantienen la seguridad, entonces podemos asumir que en ningún estado del sistema quedará comprometida.

Con mayor o menor simplicidad y elegancia los sistemas de seguridad se basan en este concepto simple para imponer la política del sistema en las operaciones ofrecidas a sus usuarios. Tecnológicamente, hay un sinnúmero de arquitecturas de implementación diferentes multiplicadas por toda la variedad de plataformas existentes.

El nivel de abstracción al que se puede implementar el monitor de referencias depende básicamente del control que el sistema tiene sobre el código ejecutado por las aplicaciones que lo componen. En plataformas interpretadas puede ser posible implementar chequeos de manera más automática, de manera externa a la aplicación (por ejemplo en el intérprete). En el caso de aplicaciones en lenguaje nativo se suele delegar la responsabilidad de implementarlos a éstas, o restringir en exceso la actividad de estas últimas de manera preventiva si no son del todo confiables.

El nivel de granularidad en la definición de la política también varía dependiendo de la arquitectura con la que se implementa el monitor de referencias. En el caso de arquitecturas que pueden proveer chequeos automáticos, la granularidad puede quedar restringida al nivel al cual se puede interceptar el código de la aplicación e identificar recursos. Algunos de estos sistemas, de todas formas,

proveen servicios de autorización a los que se le puede delegar las decisiones de control de acceso [4]. Cuando la granularidad deseada por la aplicación supera la ofrecida automáticamente por la plataforma, entonces la aplicación puede definir su propio conjunto de recursos protegidos, e implementando de manera interna el monitor de referencias, puede delegar las decisiones al servicio de autorización.

4.2 El Módulo de Control de Acceso

Con el módulo de control de acceso queremos elaborar y especificar un servicio de autorizaciones para un servidor de aplicaciones. Las aplicaciones que quieran implementar políticas de seguridad discrecionarias o obligatorias pueden utilizar el módulo delegándole las autorizaciones de control de acceso a los recursos.

No nos concentraremos en la tecnología, sino en la especificación y verificación de su funcionalidad y su lógica de funcionamiento mediante mecanismos formales. Aunque muchos aspectos tecnológicos son sin duda muy importantes, muchos serán dejados de lado en este trabajo. Uno de ellos es, por ejemplo, la tecnología que conectará al módulo con el servidor de aplicaciones y con las aplicaciones. Sólo describiremos las propiedades necesarias de esta arquitectura, definiendo las responsabilidades de cada componente.

4.2.1 Funcionalidad del módulo

El módulo de control de acceso proveerá la funcionalidad de toma de decisiones de control de acceso. Cuando una aplicación invoque al módulo para obtener un permiso de acceso sobre un recurso, este evaluará al recurso, usuario, y modo de acceso, determinando si se permite o se niega la entrega del permiso. La decisión involucrará tanto la política discrecionaria como la obligatoria, para las cuales deberá proveer una interfaz a fin de que la aplicación las pueda definir. La política discrecionaria estará representada por Listas de Control de Acceso, y la obligatoria por el modelo MLS de Bell y LaPadula. La interfaz del módulo a las aplicaciones está compuesta por los siguentes grupos funcionales:

Registrar objetos/sujetos · Mediante ciertas funciones las aplicaciones deben registrar recursos protegidos y sus usuarios.

Obtener/liberar permisos · Habrá funciones que permitirán obtener un permiso de acceso de cierto tipo sobre un objeto y luego liberarlo.

Entregar/redimir derechos discrecionarios · Existirán funciones que permitirán administrar los derechos discrecionarios de las ACLs.

Cambiar etiquetas MLS · Se permitirá cambiar las etiquetas de la política de seguridad multi-nivel, tanto de los objetos como de los sujetos.

Utilizando esta interfaz las aplicaciones pueden administrar la política de seguridad. Sin embargo, estas tienen aún responsabilidades importantes a fin de certificarla. Debido a qué no asumimos ningún marco tecnológico, no suponemos la existencia de un mecanismo que permita implementar los chequeos automáticamente. Por esto se delega la renponsabilidad, en un principio, a las aplicaciones.

La definición precisa de esta interfaz se llevará a cabo en el capítulo 5, donde se escribirá la especificación formal del módulo.

4.2.2 Responsabilidades de la aplicación cliente

Para que el control de acceso se haga efectivo, las aplicaciones tienen las siguientes responsabilidades, algunas opcionales:

Registrar recursos protegidos · Definir los recursos que desean proteger, y registrarlos. Estas operaciones sólo puede hacerse en un estado especial del sistema.

Registrar usuarios · Registrar en el módulo los usuarios sobre los que se ejerce control de acceso. Al igual que en el caso de las anteriores, sólo pueden ejecutarse en un estado especial del sistema, operado por un administrador.

Administración segura de MLS · La administración de la política de seguridad MLS debe ocurrir sólo desde un estado especial del sistema, los usuarios no deben verse involucrados.

Administración de DAC · Implementación de una interfaz para que los usuarios puedan administrar la política discrecionaria del módulo.

Completitud · Implementar la delegación al módulo de los chequeos de control de acceso para *todos* los posibles accesos a los recursos protegidos, certificando así la propiedad de *Completitud* de un monitor de referencias.

Las primeras dos son responsabilidades evidentemente básicas. La última es una responsabilidad *imprescindible* para un buen funcionamiento del módulo, ya que es parte esencial de un *monitor de referencias* (§ 4.1.2). La cuarta vale sólo si se quiere utilizar DAC. Y el resto, las cuales indican que deben ocurrir en un estado especial del sistema, son imprescindibles si se quiere implementar una política MAC (§ 2.3).

Debe prestarse especial importancia en la verificación de la implementación que las aplicaciones clientes hacen de sus obligaciones, ya que de éstas depende la correcta protección de la política de seguridad.

4.3 Conclusión

De aquí en más dejaremos de lado los aspectos tecnológicos y nos concentraremos exclusivamente en los requerimientos recién dictados. Nos disponemos a desarrollar, en el siguiente capítulo, la especificación funcional y estructural del módulo de control de acceso.

Mostraremos la especificación de su estructura interna y aquella de sus funciones principales. Luego nos focalizaremos en la especificación de las propiedades MLS deseadas, aquellas heredadas del modelo BLP.

Se aplicarán técnicas formales para diseñar una especificación rica y muy manipulable. El primer resultado favorable se mostrará en el capítulo 6 cuando se *pruebe* que la especificación tiene las propiedades deseadas.

Finalmente, en el capítulo 7, volveremos resumidamente al ámbito tecnológico con un ejemplo que mostrará una forma de aplicar la especificación desarrollada en el contexto discutido.

Capítulo 5

Especificación

En este capítulo presentamos la especificación del módulo de control de acceso. La misma se escribirá en Z, un lenguaje formal con muy buenos mecanismos de estructuración, basado en la teoría de conjuntos y la lógica matemática.

5.1 Objetivo y estructura del capítulo

Nuestro primer objetivo es desarrollar una especificación formal consistente con los requerimientos antes definidos para el módulo de control de acceso. Como hemos mencionado, implementaremos control de acceso usando ACL y el modelo de seguridad multi-nivel de Bell y LaPadula. La fusión de estos dos será similar a la desarrollada en el artículo de BLP [8], aunque esta vez con ACL.

Teniendo las especificaciones del estado del módulo, de las operaciones, y de los predicados de seguridad de interés, probaremos que las propiedades deseadas son válidas. Debido a que disponemos de un mecanismo asistido de pruebas, a través del software Z/EVES 2.1, generaremos pruebas sintácticamente puras. Como éstas no incluyen deducciones expresadas en lenguaje natural, la correción de las mismas siempre quedará certificada por el mecanismo de inferencia de Z/EVES.

La especificación en este capítulo toma la siguiente estructura:

- En la primera parte de la especificación se construyen las partes componentes del estado del módulo de control de acceso.
- En una segunda parte definiremos la política de seguridad MLS deseada en términos de como se haya definido el estado. Con esto determinaremos cuando un estado se considera contiene un compromiso de seguridad o no.
- En una tercera parte definiremos el concepto general de operación ("reglas" en BLP) como modificadora del estado del módulo de control de acceso, y veremos como lo expresamos en Z. Construiremos un marco de trabajo para poder hablar fácilmente de este concepto y así luego poder introducir las operaciones propiamente dichas de una manera simple.
- Por último, con la base de las tres primeras partes, aplicaremos la política de seguridad definida para escribir las propiedades deseadas para las operaciones.

Terminando esto abriremos el camino al próximo capítulo, en donde presentaremos los teoremas y realizaremos las pruebas. Concluiremos aquél con la prueba de un teorema importante para la concepción del módulo de control de acceso: el teorema de preservación de seguridad para una de las operaciones.

5.2 Acerca de la especificación y Z

Como ya hemos dicho, el lenguaje de especificación utilizado es Z. La herramienta principal de Z es el *esquema* [7, 15]. La especificación estará definida, en su mayoría, en términos de estos últimos. Los esquemas están especialmente diseñados para modelar *estado*. Cuentan con una sección superior en donde se definen variables tipadas y una sección inferior en donde se pueden definir predicados para restringir los valores de las variables, es decir, invariantes. En este trabajo nos estaremos refiriendo con *esquema* a la herramienta de especificación de Z a menos que se aclare lo contrario.

Una variedad de técnicas de especificación de Z se irán introduciendo a lo largo de este capítulo a medida que llegue el momento de utilizarlas.

Especificación modular para prueba claras y simples

Queremos elegir bien qué representar en cada esquema y cómo hacerlo para modularizar bien nuestra especificación y por consiguiente simplificar las pruebas.

Los beneficios de la modularización en una especificación formal, y en particular si se quieren hacer una cantidad importante de pruebas con bastantes condimentos sintácticos, son muy importantes. Así ahorramos probar repetidamente las mismas cosas, actividad que no sólo nos consume tiempo por tener que hacer las pruebas, ya sea a mano o con un asistente de pruebas, sino que también, en varias ocaciones, tenemos incluso que pensarlas de nuevo. Esto hace evidente que es fundamental identificar bien los conceptos, independizar lo más posible los esquemas, y hacer reusables a los predicados, funciones y tipos.

Se enfatiza por esto la necesidad de individualizar tanto como sea posible los conceptos utilizados. Aquellos con identidad suficiente se convertirán en esquemas con componentes e invariantes. Los más complejos tendrán teoremas que resumen sus propiedades y las propiedades de las interacciones con otros. Es a partir de la existencia de estos pequeños teoremas que los futuros serán simples y claros. Más aún con un asistente de pruebas, en donde *todo* lo utilizado debe ser probado o debe existir en el marco de trabajo del asistente. Debido a que el marco de trabajo de un asistente de pruebas suele contener sólo teoremas o propiedades de caracter muy general; nos veremos repetidas veces en la necesidad de construir teoremas pequeños que hablan sobre propiedades diversas de las entidades que vamos definiendo. Si bien al momento de su definición puede que no resulte clara su conexión con nuestro objetivo, muchas de las pruebas que realizaremos en Z/EVES requerirán bastante razonamiento que puede parecer fuera de la linea principal, aunque de hecho sea del todo necesario.

La orientación de Z hacia una buena estructuración es una gran ayuda para lidiar con los tópicos anteriores.

5.3 Especificación del estado del módulo

Especificaremos primero los conceptos que describen el *estado* del módulo de control de acceso. Casi todos fueron descritos en algún párrafo de un capítulo anterior. Comenzaremos por definir lo que para nosotros representará a los objetos y sujetos válidos del sistema. Una vez hecho esto nos concentraremos en especificar los derechos de acceso (ACL) y los permisos de acceso vigentes. Finalmente definiremos las etiquetas de seguridad relacionadas con la parte obligatoria de la política de seguridad.

5.3.1 Objetos y Sujetos

La función principal del módulo es restringir el acceso de los sujetos a los objetos del sistema, y como tal, necesita saber de la existencia de cada uno de ellos. Necesitamos definir tanto un conjunto de *sujetos* como uno de *objetos* que representen aquellos que existen y de los cuales se requiere una administración segura.

Debido a que la importancia de dichas entidades, a la vista de módulo de control de acceso, radica en poder *indentificarlas*, el resto de las características que podrían hacer a estas entidades se descartarán en esta especificación, y dichos conjuntos serán definidos como conjuntos de identificadores.

Definiremos los siguientes tipos[1]:

$UID \approx$ Identificadores de sujeto posibles.

$OID \approx$ Identificadores de objeto posibles.

Ahora podemos definir el conjunto de objetos válidos y el conjunto de sujetos válidos,

$uids \approx$ El conjunto de identificadores de sujeto en uso en el sistema.

$oids \approx$ El conjunto de identificadores de objeto en uso en el sistema.

y el primer *esquema* de la especificación:

$$| \; Administrator : UID$$

[1]En el apéndice A están las designaciones de todas las entidades de la especificación.

UsrObj
$oids : \mathbb{P}\, OID$
$uids : \mathbb{P}\, UID$

$Administrator \in uids$

El esquema *UsrObj* representa la parte del estado del módulo que contiene información básica relacionada con los sujetos y objetos, en este caso los conjuntos *uids* y *oids*. Además hemos agregado la definición axiomática $Administrator : UID$ y la línea $Administrator \in uids$ en *UsrObj*. *Administrator* representa un administrador siempre existente en el sistema, veremos el porqué de esta definición más adelante.

Z brinda un amplio conjunto de operadores que nos pueden ayudar a producir una especificación más legible y expresiva. Ejemplos de estos operadores son Δ y Ξ. Veremos más adelante cómo haber especificado el esquema *UsrObj* en forma indenendiente, separado de los esquemas que definen el resto del estado del módulo de control de acceso, nos ayuda a especlficar de una manera más conveniente el resto de las entidades de la especificación relacionadas con objetos y sujetos, mediante la utilización de estos operadores.

5.3.2 Derechos y Permisos de Acceso

Lo que sigue tiene como objetivo darnos una forma de manifestar cuándo un sujeto tiene permiso para acceder a un objeto y qué tipos de permiso tiene. En la mayoría de los sistemas de cómputo hay una variedad de formas diferentes en las que un sujeto puede acceder a un objeto, diversos "modos" de acceso. El subsistema de control de acceso debe gestionar toda la variedad de pedidos de acceso existentes para determinar si se avala o se prohibe la acción requerida.

Antes de avanzar haremos una distinción importante en este punto: necesitamos diferenciar entre el *derecho* de un sujeto para acceder a un objeto y el *permiso* de última instancia que debe pedir para acceder al mismo. Este doble control se da en especial en sistemas en donde algunos factores *circunstanciales* toman suficiente importancia como para impedir momentaneamente el acceso a un objeto por parte de un sujeto determinado. Sistemas multi-usuario concurrentes o distribuidos, en los cuales dos sujetos se pueden interferir uno al otro al intentar acceder a cierto objeto, son un ejemplo. En un principio ambos sujetos tienen el derecho de poder pedir acceso y esperar obtenerlo, pero puede que *no* puedan por razones circunstanciales que cambian de un momento a otro durante la ejecución del sistema. En nuestro caso, la necesidad radica en la separación ya explicada entre la parte dinámica y la parte estática de la política de seguridad MLS.

Hecha esta distinción, nos referiremos a *derechos de acceso* queriendo aludir a

control de acceso no circunstancial y a *permiso de acceso* cuando nos querramos referir a las autorizaciones requeridas ante un acceso inminente a un objeto.

Derechos de Acceso

Los derechos de acceso representan los permisos no circunstanciales para acceder a los objetos, que serán representados utilizando las ACLs. La selección de permisos y derechos de acceso que utilizaremos en el módulo es aquella de BLP. Cuatro de los derechos de acceso que utilizaremos se corresponden con permisos de acceso. El quinto, al igual que en BLP, no corresponde a un modo de acceso, con lo cual es un derecho de acceso pero no un permiso de acceso.

Recordemos la lista ya presentada (§ 3.2.1):

Lectura · Le permite al sujeto acceder a un objeto para extraer información.

Escritura · Pudiendo llamarse también *Adjunción*, le permite al sujeto acceder al objeto para almacenar información.

Lectura-Escritura · Le permite al sujeto acceder a un objeto tanto para extraer información como para almacenarla.

Ejecución · El sujeto puede activar un objeto.

Control · Permite alterar las propiedades externas de un objeto. No se puede modificar su contenido, aunque la información de seguridad asociada podría ser alterada, y por ende su política para control de acceso.

Como hemos dicho, los cuatro primeros se corresponden con *permisos de acceso*. Esto quiere decir que existirán permisos de acceso LECTURA, ESCRITURA, LECTURA-ESCRITURA y EJECUCIÓN. La posibilidad de obtención de estos "modos de acceso" estará determinada por la posesión del correspondiente *permiso*. Qué hecho indicará que un sujeto *posee* un permiso será modelado más adelante.

Con respecto al último derecho de acceso (CONTROL), alude al conjunto de acciones de carácter más administrativo sobre los objetos. Acciones administrativas en la implementación de un sistema que utilize el módulo podrán requerir la posesión de este derecho de acceso para poder efectuarse. Un ejemplo típico son aquellas operaciones relacionadas con la seguridad de los objetos, o con su creación o eliminación. Este tipo de permiso es poseido en la mayoría de los casos solo por el dueño del objeto.

El derecho de acceso CONTROL se obtiene en la creación de un objeto, aunque también puede obtenerse por otros medios en circunstancias especiales. Lo más común será que sólo los dueños de cierto objeto tengan derecho de CONTROL sobre el mismo. En todos los casos, este derecho de acceso se obtiene sólo en un

estado especial del sistema, a través de un usuario de privilegios especiales. La administración de CONTROL es parte de la política obligatoria, no discrecionaria, debido a que no es recomendable dejar demasiada libertad a la propagación de derechos de acceso discrecionarios [12]. De esta manera, sólo se deberá permitir administrar CONTROL a los *administradores de la política de seguridad obligatoria.*

Definiremos primero el conjunto de modos de acceso disponibles en el sistema:

$$\begin{aligned} ACCESS\_ATTRIBUTE ::= \; & Read \\ | \; & Write \\ | \; & ReadWrite \\ | \; & Execute \\ | \; & Control \end{aligned}$$

El conjunto de atributos de acceso del sistema define la mínima granularidad con que se distingue entre las diversas formas Todos ellos son a su vez derechos de acceso, por lo que este último conjunto es definido de la siguiente manera:

$$RIGHT == ACCESS\_ATTRIBUTE$$

Utilizaremos principalmente los derechos de acceso para administrar la parte discrecionaria de nuestra política de seguridad mixta.

Permisos de Acceso

En contraposición a los derechos de acceso, es necesario obtener un *permiso de acceso* en un punto inmediatamente previo a ejecutar la acción que lo requiere.

Se debe notar que el término *permiso de acceso* alude a lo que en BLP es un "modo de acceso". La *no* distinción entre un "modo" y un *permiso de acceso*, que en un principio podría pensarse como la misma cosa, puede llevar confusiones conceptuales. El primero es de caracter más técnico y alude a prever qué tipo de operaciones se harán con el objeto, también suele utilizarse para control de concurrencia. El segundo determinará que el sujeto está *habilitado* para acceder a cierto objeto, pudiendo ser o no un hecho que ha pedido el "modo de acceso" necesario. En un principio, no necesariamente la obtención de uno u otro, en un entorno en el cual ambos se conjugan, debe darse al mismo tiempo. Es por eso que hacemos la distinción y *no* utilizaremos el término "modo de acceso" utilizado por BLP.

Un requerimiento por un permiso de acceso debe ocurrir en cualquier instante desde la ejecución de la primera sentencia de un programa hasta el momento anterior a la acción que lo necesita. Cabe notar que en contexto de control de acceso la obtención de cierto permiso no puede estorbar a otros usuarios, como

en el caso de los "modos de acceso" en los sistemas operativos. Sólo restringe al *mismo* usuario a través de la propiedad de confinamiento (§ 2.3.2).

El conjunto de permisos de acceso también es subconjunto del ya definido *ACCESS_ATTRIBUTE*. Lo denominaremos *PERMISSION*, y se define como sigue:

$$PERMISSION == ACCESS\_ATTRIBUTE \setminus \{Control\}$$

En Z el símbolo \ representa la diferencia de conjuntos. Notar que hemos tratado como a un conjunto al tipo *ACCESS_ATTRIBUTE* al restarle $\{Control\}$, cuando en realidad fue definido como un *tipo libre*. En Z, si un *tipo libre* es tratado como conjunto éste se valua como el conjunto de sus términos [7, 15].

5.3.3 Permisos de Acceso vigentes

Para expresar que cierto sujeto tiene un permiso de acceso dado sobre un objeto definiremos una función. Ésta tomará un elemento del producto cartesiando entre sujetos y objetos válidos ($uids \times oids$), y nos dará un subconjunto de *PERMISSION* que representará el conjunto de permisos de acceso que posee el sujeto sobre el objeto.

La función, a la que llamaremos *permSet*[2], se conviertirá así en parte del estado del módulo, ya que define para un estado dado quiénes potencialmente pueden acceder a qué recursos. Como tal, habrá una manera de modificarla para mantener una representación correcta de los permisos vigentes. Asumiremos que cuando un permiso de acceso aparece en la imagen de *permSet* sobre un par (sujeto, objeto) el sujeto está *pontencialmente* accediendo al objeto en el modo de acceso asociado. Los sujetos deben pedir un permiso de acceso para acceder a un objeto, pero esto no implica que cuando lo tengan hagan uso de él. De hecho un sujeto podría pedir un permiso de acceso para después liberarlo sin finalmente utilizarlo. Análogamente, cuando uno no aparece, asumiremos que el sujeto *no* está accediendo al objeto en dicho modo.

Recordemos que todo cliente del módulo debe delegar al módulo *toda* decisión acerca de la gestión de accesos (§ 4.2.2). Y es su responsabilidad la *inaccesibilidad* de todo objeto por parte de los sujetos que no tienen el permiso correspondiente. En este hecho basamos la integridad de la política de seguridad. El mecanismo por el cual se certifica depende de esto, ya que expresaremos el concepto de *compromiso de seguridad* en términos de *permSet*. Esta función será la única percepción del módulo sobre los accesos que pueden estar ocurriendo en el sistema.

[2] La funcion *permSet* es análoga a la matriz b de BLP.

Definimos entonces la función *permSet*, la representación de los permisos de acceso vigentes, como sigue:

$$UOp == UID \times OID$$
$$fUO\_PERMS == UOp \pfun \mathbb{P}\, PERMISSION$$
$$permSet : fUO\_PERMS$$

Se ha utilizado, y para esto definido, la abreviatura fUO_PERMS, y en segunda instancia UOp. Destacaremos también la utilización del símbolo $\pfun$, que representa en Z una función parcial. Esto significa, para nuestro caso, que no siempre una función de tipo fUO_PERMS será aplicable sobre un elemento del conjunto UOp. Nos referiremos con el término *dominio* al conjunto de elementos sobre los que la función es aplicable. Su tipo solo determina el tipo de los elementos de su dominio, no lo representa. En este caso el dominio de la función estará contenido en UOp pero no necesariamente abarcará todo el espectro del tipo. Discutiremos un aspecto importante sobre el dominio de *permSet* en la próxima sección.

Definimos estas abreviaturas (fUO_PERMS, UOp) porque usaremos mucho estos tipos. En particular, en las diversas propiedades que aluden a los permiso de acceso y sus correspondientes pruebas. Es más fácil razonar sobre expresiones más cortas y estructuras más simples.

La función *permSet* en la operación del módulo

Cuando un sujeto pida un nuevo permiso de acceso, el elemento correspondiente de *PERMISSION* será agregado a la imagen del par (sujeto, objeto) por *permSet*. Cuando un sujeto libere un permiso de acceso de su posesión, se eliminará dicho permiso de su imagen respecto de la misma función.

Participarán en el dominio de *permSet* sólo los pares con sujetos y objetos válidos, es decir, aquellos en $uids \times oids$. Otra opción podría ser que su dominio sea todo $UID \times OID$, tal como se hizo en el artículo de BLP. En este caso la función debería definir un comportamiento arbitrario para los casos que no tienen sentido. Por ejemplo, podría retornar el conjunto vacío para todo par inválido (alguna componente, o el sujeto o el objeto, es inválida). Podríamos pasarle a *permSet* dos identificadores inválidos para el sistema y ésta seguiría retornando un conjunto vacío. Si bien la última opción parece más simple de implementar (y en muchas casos esta es la forma en que se programa), no nos ayuda a producir un programa correcto. Definir funcionalidades lo más acotadas posible a los requerimientos nos ayuda a producir mejores verificaciones y a menudo más simples. Debido a nuestra elección nos veremos obligados a actualizar el dominio de *permSet* cuando nuevos objetos o sujetos se registren o cuando se eliminen. Sin embargo, este hecho no es difícil de especificar en

Z, ya que dispone de operadores especiales para operar con el dominio de las funciones y para sobreescribirlas.

Inicializaremos *permSet* para todo par (sujeto, objeto) dentro de su dominio con el conjunto vacío.

Definición de un esquema para los permisos de acceso

Ahora definiremos un esquema que encapsulará los datos recién descritos. El esquema *ActualPermSet* representa la parte del estado del módulo de control de acceso que contiene la información relacionada con los permisos de acceso.

ActualPermSet

$$UsrObj$$
$$permSet : fUO\_PERMS$$

$$\text{dom } permSet = uids \times oids$$

Notar que hemos utilizado por primera vez la segunda sección de una definición de esquema (la sección inferior). Como dijimos antes, nos permite definir invariantes de estado: restricciones a los valores de las variables definidas en la primera sección mediante predicados que deben cumplirse siempre. Allí definimos el dominio de la función *permSet*.

También hemos utilizado otra técnica común en Z, la *macroexpansión*. La macroexpansión consiste en incluir un esquema en la parte de declaraciones un nuevo esquema para que éste herede las declaraciones y los predicados del anterior. Se expresa incluyendo el nombre del esquema en un renglón de la parte declarativa. Este mecanismo permite la modularización de esquemas mediante composición, aunque, a diferencia de otros tipos de composición, el espacio de nombres de las variables es compartido. Así, es posible referenciar aquellas variables definidas en el esquema incluido en forma directa, sin indicar a que esquema pertenecen originalmente. Este es el caso de *uids* y *oids*, que pertenecen a *UsrObj*. Las entidades con los mismos nombres se fusionan, y en caso de no ser posible (por diferencias de tipo) la macroexpansión no será valida. Sólo sería posible utilizarla aplicando renombre de variables.

5.3.4 Derechos de Acceso

Pasamos ahora a focalizarnos en los derechos de acceso. En forma análoga a la sección anterior, definiremos el esquema *ACLS* que representa el estado de la política discrecionaria, representada por las ACLs.

Como hemos visto, una ACL tiene la estructura de una lista de pares (sujeto, permiso de acceso). Estos últimos definen la política de seguridad discrecio-

naria para un objeto dado. Se tiene una ACL asociada a cada objeto.

Definiremos en el módulo la función *acl* que dado un objeto nos retornará un conjunto de pares (sujeto, permiso de acceso) que representará los sujetos que pueden acceder al objeto y en qué modos pueden hacerlo. Es decir, un sujeto dado solo podrá acceder a un objeto si el par asociado al permiso deseado pertenece al conjunto de pares que es imagen del objeto mediante la función *acl*. La definición de *ACLS* sigue:

ACLS

$$UsrObj$$
$$acl : OID \pfun \mathbb{P}(UID \times RIGHT)$$

$$\mathrm{dom}\ acl = oids$$

Cada operación que desee alterar los derechos de acceso de un objeto ejecutará instrucciones para alterar la función *acl*[3] y por consiguiente la política discrecionaria del módulo.

5.3.5 Clases de Seguridad

La porción del estado del módulo que contiene los datos necesarios para implementar la parte obligatoria de la política de seguridad involucra la asignación de etiquetas (o *clases*) de seguridad a los elementos del sistema.

Hemos visto ya la estructura de las clases de seguridad que utiliza BLP, a su vez basada en la de la política del DoD. Las mismas están compuestas por un *nivel* y por un conjunto de *categorías*. Así, especificaremos una clase de seguridad con un esquema constituido de sus dos componentes:

SecClass

$$level : LEVEL$$
$$categories : \mathbb{P}\ CATEGORY$$

$LEVEL \approx$ Niveles posibles para las etiquetas de seguridad.

$CATEGORY \approx$ Categorías posibles para las etiquetas de seguridad.

Hemos introducido los tipos correspondientes a los niveles y a las categorías. Los valores involucrados en estos conjuntos se mantienen genéricos. Sólo se requiere que el conjunto *LEVEL* esté totalmente ordenado y sea, naturalmente, finito. El orden aquí será expresado con los símbolos de orden estandar ($<$, $\geq$, etc.), asumiremos que estos definen un orden total en *LEVEL*. Por ser finito

[3] La función *acl* es análoga a la matriz M de BLP.

además este orden tiene un primer y último elemento, los que referenciaremos con $\min(LEVEL)$ y $\max(LEVEL)$ respectivamente.

Ahora que tenemos la definición de una clase de seguridad debemos etiquetar con éstas a todos los sujetos y objetos en el sistema. Cada elemento del sistema deberá tener asociada una y sólo una etiqueta de seguridad: *su* clase de seguridad.

Como ya lo hicimos en las secciones anteriores, expresaremos este tipo de relaciones como una función parcial. Definiremos ahora el tipo de las funciones que necesitamos y luego el esquema *SecClasses*:

$$fUC == UID \pfun SecClass$$
$$fOC == OID \pfun SecClass$$

SecClasses

$$UsrObj$$
$$usrclass : fUC$$
$$objclass : fOC$$

$$\mathrm{dom}\ usrclass = uids$$
$$\mathrm{dom}\ objclass = oids$$

Nuevamente hemos definido estas funciones como parciales ($\pfun$). También como antes, hemos definido invariantes para definir el dominio de las funciones *usrclass* y *objclass*; e incluido el esquema *UsrObj* usando macroexpansión para tener acceso los conjuntos de objetos y sujetos válidos.

5.3.6 Adminstración de MAC

De acuerdo a lo ya mencionado, la administración de las etiquetas de seguridad, la creación y eliminación de objetos, y la asignación de *Control* sólo puede llevarla a cabo un usuario con permisos especiales de administración de la política de seguridad obligatoria.

Esto lo modelaremos con un esquema diseñado para este fin. Definiremos un conjunto de usuarios, así como ya de definió el conjunto *uids*, que represente aquellos usuarios que cuentan con estos permisos especiales. Definiremos el esquema *MACAdmin* como sigue:

MACAdmin

$$UsrObj$$
$$macAdmins : UID$$

$$macAdmins \subseteq uids$$
$$macAdmins \neq \emptyset$$

Sólo los usuarios presentes en el conjunto *macAdmins* podrán ejecutar las operaciones que alteran la política de seguridad mandatoria.

5.3.7 Estado completo del módulo

Hemos terminado de especificar las variables de estado necesarias para el funcionamiento de módulo.

El diseño modularizado permitirá especificar más limpiamente los predicados y el comportamiento de las funciones que alteran variables especificas del estado del módulo. Sin embargo, podemos querer definir propiedades que involucran a todas las variables de estado del módulo; por ejemplo: "nada ha cambiado". Por esta razón definiremos un esquema que combina los esquemas anteriores en un único esquema en el que se macroexpanden sus variables y predicados. Aprovecharemos esta oportunidad para introducir, además, ciertos operadores para esquemas que utilizaremos bastante.

$$SystemState \mathrel{\widehat=} UsrObj \land SecClasses \land ACLS \land ActualPermSet$$

El símbolo $\widehat=$ define un nuevo esquema utilizando una expresión especial. La expresión a la derecha de la definición está constituida por esquemas y operadores entre esquemas, en este caso sólo $\land$. El operador entre esquemas $\land$ construye un esquema a partir de otros dos, y su semántica es análoga a aquella de la macroexpansión (§ 5.3.3).

Con esto podemos expresar, por ejemplo, que luego de la ejecución de una operación OP nada debe haber cambiado en el estado del módulo de control de acceso:

$$OP \Rightarrow \Xi SystemState$$

Otro símbolo que se introduce aquí es Ξ, que indica que el esquema asociado no es alterado por la operación (aunque esta es una explicación algo simplificada).

5.3.8 Estado Inicial

Hemos definido todas las componentes del estado del módulo pero falta aún indicar cual es el estado inicial. Es importante verificar que este estado es seguro respecto de la política de seguridad definida.

Debido a que sólo sujetos autorizados pueden registrar usuarios en el módulo, asumiremos la existencia de un usuario especial llamado *Administrator* (§ 5.3.1) que siempre existirá en el sistema. De esta forma será posible agregar inicialmente los usuarios deseados, ya que de no existir no habría ningún usuario que pudiera ejecutar las operaciones requeridas.

InitialState

$$
\begin{array}{|l}
\hline
SystemState \\
\hline
oids = \emptyset \\
uids = \{Administrator\} \\
macAdmins = \{Administrator\} \\
permSet = \{(Administrator \mapsto \emptyset)\} \\
acl = \emptyset \\
usrclass = \{(Administrator \mapsto TopSecClass)\} \\
objclass = \emptyset \\
\hline
\end{array}
$$

Asignaremos al usuario *Administrator* la clase de seguridad máxima en el sistema, *TopSecClass*, así asegurándonos que siempre habrá alguien capaz de administrar la entidades de más alta clasificación. Definimos *TopSecClass* de la siguiente manera:

TopSecClass

$$
\begin{array}{|l}
\hline
SecClass \\
\hline
categories = CATEGORY \\
level = \max(LEVEL) \\
\hline
\end{array}
$$

Aunque no es del todo relevante para las tareas de específicas de administrador (registrar/eliminar objetos y sujetos, y cambiar las clases de seguridad), en favor de futuras extensiones al trabajo ésta es la clase de acceso más adecuada para esta entidad.

5.4 Especificación de la política de seguridad

Con el estado completo, comenzaremos a especificar la política de seguridad. La política del módulo de control de acceso es una política mixta, con aspectos discrecionarios para la administración fina y aspectos obligatorios para una gestión global de la información. Adicionalmente, los controles obligatorios sirven como mecanismo para combatir los ataques por caballos de Troya.

Hemos visto en el capítulo 3 que BLP define dos predicados que deben validarse a fin de certificar la seguridad del sistema, así describiendo formalmente su política:

- La propiedad de seguridad simple
- La propiedad-★

Ambas propiedades se definen sobre el orden impuesto al conjunto de etiquetas. Para facilitar la escritura de dichas propiedades, definiremos antes dos relaciones

importantes en las cuales se basan. Ambas son relaciones análogas a relaciones definidas en BLP. Primero definiremos la relación *dominates*, que corresponde al orden parcial del conjunto de etiquetas. Seguido daremos la definición de *secureRel*, relación entre el conjunto de pares $UOp \times PERMISSION$ y el conjunto de estados del sistema en los cuales el permiso de acceso asociado al par no compromete la seguridad.

5.4.1 La relaciones *dominates* y *matches*

Definiremos el orden en el conjunto *SecClass* con la siguiente relación:

syntax *dominates inrel*

$$\_dominates\_ : SecClass \leftrightarrow SecClass$$

$$\forall x, y : SecClass \bullet$$
$$x \ dominates \ y \Leftrightarrow$$
$$(y.level \leq x.level \land y.categories \subseteq x.categories)$$

La relación *dominates* define el orden parcial que utilizaremos para las etiquetas de seguridad. Corresponde al estilo de orden parcial utilizado en la política del DoD que hemos analizado previamente. Naturalmente, en esta especificación toma la forma de una relación entre dos entidades de tipo *SecClass*.

Notar que como usaremos *dominates* en forma infija, necesitamos usar la instrucción *syntax* para indicarle Z/EVES que no intente compilar y chequear esta sintaxis como si fuera una aplicación de funciones.

Definiremos también la relación *matches* que representa la equivalencia entre clases de seguridad. De manera similar a *dominates*, la definición sigue:

syntax *matches inrel*

$$\_matches\_ : SecClass \leftrightarrow SecClass$$

$$\forall x, y : SecClass \bullet$$
$$x \ matches \ y \Leftrightarrow$$
$$(y.level = x.level \land y.categories = x.categories)$$

Notar que (x *matches* y) equivale a (x *dominates* y) $\land$ (y *dominates* x).

5.4.2 La relación *secureRel*

Definiremos ahora una relación de alto nivel que mejora nuestra expresividad sobre la política de seguridad. Basandonos en la definición de *dominates*, definiremos la relación *secureRel*. La misma simplifica los predicados que luego hablarán sobre la existencia de un compromiso de seguridad a nivel del sistema.

La relación *secureRel* determina que cierto terna (sujeto, objeto, permiso), que representa un acceso a un recurso por parte de un sujeto, no genera un compromiso de seguridad si el mismo es otorgado en el contexto de cierto estado del sistema.

Con esta idea, la definición sigue así:

syntax *secureRel inrel*

$$
\begin{array}{|l}
\_secureRel\_ : UOp \times PERMISSION \leftrightarrow SecClasses \\
\hline
\forall f : SecClasses \bullet \\
\quad \forall u : f.uids;\ o : f.oids;\ a : PERMISSION \bullet \\
\quad\quad ((u, o), a)\ secureRel\ f \Leftrightarrow \\
\quad\quad\quad (a \in \{Read, ReadWrite\} \Rightarrow \\
\quad\quad\quad\quad f.usrclass\ u\ dominates\ f.objclass\ o)
\end{array}
$$

Esta relación se define de tal forma que solo queden relacionados aquellos pares en donde el permiso de acceso a o bien no es de lectura, o si es de lectura (a saber, *Read* o *ReadWrite*) entonces el objeto o tiene una clase que es dominada por la clase que posee el sujeto u. El contexto es cierto valuación para el esquema *SecClasses* (§ 5.3.5), el lado derecho de la relación.

5.4.3 Propiedad de Seguridad Simple

Ahora estamos en condiciones de definir el concepto de estado seguro respecto de nuestra política obligatoria. Este corresponde al concepto seguridad simple.

El sistema se encuentra en un estado seguro si no tiene un compromiso de seguridad. Es decir, ningún sujeto esta accediendo a información cuya clase de seguridad no domina, y de tal manera la definición sigue:

$$
\begin{array}{|l}
\text{SimpleSecureState} \\
\hline
ActualPermSet \\
SecClasses \\
\hline
\forall p : UOp;\ a : PERMISSION \mid \\
\quad p \in uids \times oids \wedge a \in permSet\ p \\
\quad \bullet\ (p, a)\ secureRel\ \theta SecClasses \\
\hline
\end{array}
$$

Si un estado del sistema valida el predicado anterior, entonces el principio de seguridad simple se cumple. Como hemos explicado antes, este concepto representa la forma más esencial de la política de seguridad.

5.4.4 Propiedad-★

Definiremos ahora la propiedad de confinamiento o propiedad-★, que tiene una forma un poco más compleja. Recordemos que ésta dice lo siguiente:

la clase de seguridad de un objeto que este siendo leído por un sujeto no supera la clase de seguridad de un objeto que el mismo sujeto esté escribiendo

Definición de la función objSAtts

Antes de escribir la propiedad-★ definiremos una función que no solo será de ayuda sintáctica sino que también permitirá aislar varias propiedades generales (y sus pruebas). Sigue la definición de la función *objSAtts*:

$$
\begin{array}{|l}
objSAtts : ActualPermSet \times UID \times \mathbb{P}\, PERMISSION \nrightarrow \mathbb{P}\, OID \\
\hline
\mathrm{dom}\, objSAtts = \{m : ActualPermSet;\ u : UID; \\
\qquad atts : \mathbb{P}\, PERMISSION \mid u \in m.uids\} \\
\forall\, m : ActualPermSet \bullet \\
\quad \forall\, u : m.uids;\ atts : \mathbb{P}\, PERMISSION \bullet \\
\qquad objSAtts\ (m, u, atts) \subseteq m.oids \\
\forall\, m : ActualPermSet \bullet \\
\quad \forall\, u : m.uids;\ atts : \mathbb{P}\, PERMISSION \bullet \\
\qquad objSAtts\ (m, u, atts) = \\
\qquad\qquad \{o : m.oids;\ a : atts \mid a \in m.permSet\ (u, o) \bullet o\}
\end{array}
$$

La función *objSAtts*[4] se aplica sobre un estado del módulo (en realidad una parte), un usuario y un conjunto de permisos de acceso; y retorna aquellos objetos del sistema para los que el sujeto tiene alguno de los permisos enumerados.

Notar que la definición de *objSAtts* esta regida por tres predicados que definen su dominio y comportamiento. El primero corresponde a la definición del dominio sobre el cual es posible aplicarla. La función es parcial y no se puede aplicar sobre todo el espectro de su tipo. La única restricción de dominio para esta función es que debe ser aplicada sólo sobre aquellos identificadores de sujeto que estén en *uids*. El segundo refina el rango de la función, certificando que todos conjuntos de los identificadores de objeto que son valuaciones de la misma están en el conjunto *oids*. El último y tercer predicado, finalmente, define su comportamiento antes descrito.

Definición de *StarPropertyState*

En analogía a *SimpleSecureState* definimos el esquema *StarPropertyState*, que representa el conjunto de los estados que satisfacen la propiedad-★:

[4] *objSAtts* equivale a la notación $b(S : w, a, \ldots)$ usada por BLP.

StarPropertyState

$$
\begin{array}{l}
ActualPermSet \\
SecClasses \\
\hline
\forall u : UID;\ ow, or : OID \mid u \in uids \land \\
\quad ow \in objSAtts\ (\theta ActualPermSet, u, \{Write, ReadWrite\}) \land \\
\quad or \in objSAtts\ (\theta ActualPermSet, u, \{Read, ReadWrite\}) \\
\quad \bullet\ objclass\ ow\ dominates\ objclass\ or
\end{array}
$$

Notar que cuando escribimos un esquema en representación de una propiedad interesante, su definición incluye sólo las partes del estado de las que necesita hablar. Para utilizarlo, conjugamos el esquema definido con un estado cualquiera mediante un conector lógico. El siguiente es un ejemplo:

$$\exists E : SystemState \bullet E \land StarPropertyState$$

Este predicado afirma que existe un estado E tal que la propiedad-★ se cumple. Válido, por ejemplo, para un estado sin accesos. El primer estado que deseamos valide estos predicados es *InitialState*:

$$InitialState \Rightarrow StarPropertyState \land SimpleSecureState$$

5.5 Especificación de la operaciones

Habiendo definido los conceptos de estado y estado seguro, nos disponemos a especificar las operaciones. La funcionalidad del módulo de control de acceso queda definida por tales, siendo la única interfaz al usuario de este último.

5.5.1 Interfaz del módulo

Las operaciones a definir para el módulo, según lo descrito en la sección 4.2, pueden ser agrupadas en cuatro secciones.

La primera, que alude a la registración de sujetos y objeto en el módulo, dispondrá de cuatro funciones:

createObject(oid:OID) · Registra un objeto en el módulo de control de acceso con un *oid* dado. El *oid* no debe estar siendo utilizado para referenciar a ningún objeto al momento de la invocación.

deleteObject(oid:OID) · Elimina un objeto del conjunto de objetos gestionados por el módulo de control acceso. El parámetro *oid* indíca el objeto a desregistrar.

createSubject(uid:UID) · Registra un sujeto en el módulo de control de acceso con un *uid* dado. Al igual que en el caso de los objetos el *uid* no debe estar utilizado para referenciar a ningún otro sujeto.

deleteSubject(uid:UID) · Elimina un sujeto del módulo. El parámetro *uid* indica el sujeto a desregistrar.

Hay que tener en cuenta que hemos asumido, como ya fue explicado, que estas funciones sólo se invocan en un estado especial del sistema si se quiere implementar correctamente una política MAC. Esto se da también para el siguiente grupo de funciones, que permiten cambiar las etiquetas de seguridad MLS de los objetos y sujetos:

changeObjSecClass(oid:OID, class:SecClass) · Cambia la clase de un objeto gestionado por el módulo. El parámetro *oid* es el identificador del objeto al que se le quiere cambiar la clase y el parámetro *class* es la nueva etiqueta de seguridad. El sujeto que invoca la operación debe tener permisos de administración MAC.

changeUsrSecClass(uid:UID, class:SecClass) · Cambia la clase de un sujeto gestionado por el módulo. Equivale a la anterior pero en analogía para un sujeto.

También deben especificarse las funciones para administrar la política de seguridad discrecionaria de ACL:

give(uid:UID, uid2:UID, oid:OID, mode:PERMISSION) · Entrega a un sujeto un derecho de acceso dado.

rescind(uid:UID, uid2:UID, oid:OID, mode:PERMISSION) · Le quita a un sujeto cierto derecho de acceso sobre un objeto.

Por último, están las funciones que implementan la gestión de permisos de acceso:

getRead(uid:UID, oid:OID) · Entrega a un sujeto el permiso de acceso de lectura sobre cierto objeto.

getWrite(uid:UID, oid:OID) · Entrega el permiso de acceso de escritura sobre cierto objeto.

getReadWrite(uid:UID, oid:OID) · Entrega a un sujeto el permiso de acceso de lectura-escritura sobre cierto objeto.

getExecute(uid:UID, oid:OID) · Entrega a un sujeto el permiso de acceso de ejecución.

release(uid:UID, oid:OID, mode:PERMISSION) · Libera un permiso de acceso entregado anteriormente.

Las siguientes secciones dan la especificación formal en Z para cada una de las operaciones enumeradas. Dado que constituyen la única manera de modificar el estado del módulo, tienen la responsabilidad de tener que dejar al sistema en consistencia con la política de seguridad.

Tal responsabilidad, dado que el primer objetivo del módulo es la seguridad, implica serias verificaciones a fin de certificar su correcta definición e implemementación. Deseamos de nuestro sistema que para cada estado de su ejecución se satisfagan los predicados de seguridad *SimpleSecureState* y *StarPropertyState*. Por lo tanto las operaciones no deben alterar este invariante. Es decir, si una operación se ejecuta, como es debido, sobre un estado *seguro* del sistema, entonces debe alterarlo de tal forma que esta condición siga valiendo en el estado resultante. Partiendo de un estado inicial del sistema seguro, hace falta probar sólo el predicado anterior para certificar que no puede introducirce un compromiso de seguridad en el sistema a través de las operaciones.

Con esto en mente, especificaremos el comportamiento de las funciones principales del módulo.

5.5.2 Definiciones auxiliares

Procediendo de manera similar a cuando se definieron los predicados de la política de seguridad, en esta sección haremos varias definiciones que nos ayudarán a definir las operaciones. Esto incluye funciones que mejoran la claridad sintáctica y esquemas generales que reusaremos a lo largo de las operaciones mediante macroexpansión o lógica de esquemas.

Construcción de casos de error

Comenzaremos definiendo un esquema que representa un caso excepcional en la ejecución de una operación, tal es el caso de la inexistencia de cierto elemento del sistema al que la llamada a la operación hace referencia, p.ej. un identificador de objeto inválido.

Cada operación deberá informar mediante una variable de salida, un valor de retorno, o una excepción (dependerá de la implementación), si ha tenido éxito o no. Modelaremos esto con el tipo *DECISION* correspondiente a una variable de salida *out* y al concepto *decision* ya conocido dentro del modelo BLP [8]. La

variable *out* aparecerá en todas las operaciones para indicar si hubo un error de ejecución o no.

En general tendremos que las operaciones manifestan alguno de tres casos de ejecución diferentes:

Ejecución satisfactoria · Las guardas permiten que la operación se ejecute con naturalidad

Denegada por la política de seguridad · Las guardas no perminten que se ejecute la operación pero no por un error sinó por una decisión relacionada con la política de seguridad.

Error de ejecución · La operación no puede completarse a causa de una guarda que la proteje de un error de ejecución.

Para construir los diversos errores que pueden ocurrir en la llamada a una operación del módulo de control de acceso definiremos el concepto de excepción: un par cuya primera componente es un identificador del tipo de error y cuya segunda componente un identificador de la función en la cual ocurrió la excepión. Lo definiremos de la siguiente manera:

$FUNCTION \approx$ El conjunto de todos los identificadores de funciones.

$MESSAGE \approx$ Mensajes del sistema (en este caso todos errores, podría haber otros).

$$EXCEPTION == MESSAGE \times FUNCTION$$

Luego, las definiciones de *DECISION* y del esquema *Error* siguen:

$$DECISION ::= YES \mid NO \mid mkerr \langle\langle EXCEPTION \rangle\rangle$$

Error

$function : FUNCTION$
$message : MESSAGE$
$out! : DECISION$

$out! = mkerr\ (message, function)$

El esquema *Error* representa un error de ejecución durante la ejecución de una operación del módulo de control de acceso, y su definición tiene el fin de generalizar la definición de estos casos.

Definiremos también esquemas para los dos casos de errores más comunes en el conjunto de operaciones: una referencia a un sujeto inválido (que no pertenece a

uids) y una referencia a un objeto inválido (que no pertenece a *oids*). Haremos esto creando dos esquemas que utilizan la macroexpansión de *Error* con el parametro *message* instanciado para el valor particular correspondiente al caso.

La definición de esquema que representa el caso de error causado por un objeto inexistente sigue:

ErrorObjDoesNotExist

$$
\begin{array}{|l}
UsrObj \\
o? : OID \\
u? : UID \\
Error[message := ObjDoesNotExist] \\
\hline
o? \notin oids \land u? \in uids \\
\hline
\end{array}
$$

Notar que se ha instanciado la variable *message* del esquema *Error* con el valor *ObjDoesNotExist*, y se ha dispuesto la condición para la ocurrencia del error en la sección de invariantes. En particular, este caso se da cuando el sujeto $u?$ está disponible para operar pero referencia a un objeto que no está en *oids*.

De manera similar, la definición de *ErrorInvalidUser* sique así:

ErrorInvalidUser

$$
\begin{array}{|l}
UsrObj \\
u? : UID \\
Error[message := InvalidUser] \\
\hline
u? \notin uids \\
\hline
\end{array}
$$

Hemos definido este error de tal forma que captura más casos que el análogo para objetos, ya que si ambos casos se dan, entonces éste prevalece y es informado y el segundo no. Si $u?$ no está en *uids*, aunque $o?$ no pertenezca a *oids* el error informado será el correspondiente a $u?$.

Utilizaremos los esquemas representativos de los casos de error de la siguiente manera:

$$CasoErrorOperacion1 \mathrel{\widehat{=}} CasoError[function := idOperacion1]$$

Donde *CasoErrorOperacion*1 es el esquema que representa el caso de error a modelar de la operación, *CasoError* es uno de los esquemas recién definidos (a los que sólo les falta instanciar la variable *function*), e *idOperacion*1 es el identificador de la operación.

Un esquema para la entrega de permisos de acceso

Como varias de las operaciones tienen por objeto entregar cierto permiso de acceso a un objeto, y de hecho son las más relevantes al módulo de control de

acceso; generalizaremos este concepto especificando un esquema que representa una operación que entrega un nuevo permiso de acceso a un sujeto.

El esquema *AugmentingAccess* representa una operación que toma por parámetro un sujeto, un objeto, y un permiso de acceso y resulta en agregar el modo de acceso correspondiente a la función *permSet*. La definiremos de la siguiente manera:

AugmentingAccess

$\Delta ActualPermSet$
$a : PERMISSION$
$u : UID$
$o : OID$

$(u, o) \in \text{dom } permSet$
$permSet' = augb\ (permSet, (u, o), a)$

Definiremos la función *augb* en la siguiente sección. El objetivo de esta función es agregar el modo de acceso a al sujeto u sobre el objeto o. Con este esquema podemos construir, análogamente a lo que fue mostrado para *Error*, operaciones donde parte del comportamiento ya ha sido especificado en este esquema general. Veremos su utilización directamente en la definición de las operaciones.

La función *augb*

Definiremos la función *augb* como parte del nuestro marco de trabajo para poder expresar fácilmente cuándo queremos agregar un nuevo permiso de acceso al estado del módulo de control de acceso. La función hereda el nombre de *augb* de la utilizada en el artículo de BLP [8].

La aplicación $augb\ (permSet, (u, o), a)$ representa la *función* que equivale a *permSet* salvo en (u, o), donde su imagen incluye la imagen de *permSet* pero además contiene el permiso de acceso a.

$$augb : fUO\_PERMS \times UOp \times PERMISSION \nrightarrow fUO\_PERMS$$

$$\text{dom } augb = \{f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid up \in \text{dom } f\}$$
$$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid up \in \text{dom } f \bullet augb\ (f, up, na) = f \oplus \{(up \mapsto fup \cup \{na\})\}$$

El símbolo $\oplus$ es un operador que en Z se aplica entre funciones. La expresión $f \oplus g$ evalua a una función que se comporta como f excepto cuando el elemento sobre el que se aplica pertenece al dominio de g, en donde se comporta como g. Se puede entender como una *sobreescritura*. El término $(a \mapsto b)$ es equivalente a (a, b). En Z las funciones son conjunto de pares.

También hay una contrapartida de *augb* que, en vez de agregar un nuevo permiso de acceso, quita uno. Pero esta función, la cual se denomina *dimb*, será definida cuando tengamos que definir la operación que libera un modo de acceso (§ 5.5.7).

Estructura de una definición de operación

Toda operación debe ser aplicable a cualquier *SystemState* y retornar una *DECISION* en respuesta. Cada vez que incluimos una precondición en la especificación de cierta parte de una operación necesitamos incluir los casos excluidos en alguna otra parte especificada. Luego se combinan las partes definidas para formar una operación aplicable sobre cualquier estado. La estructura del tipo *DECISION* nos define así la forma en que se definen las operaciones:

$$Op \mathrel{\widehat{=}} OpYes \lor OpNo \lor OpEr_1 \lor \ldots \lor OpEr_n$$

Por conveniencia generalmente agruparemos los casos de error y el de rechazo (*mkerr e* y *NO*) en un único esquema, y sacaremos el estado común no afectado fuera de la especificación de los casos por simplicidad. Una especificación de una operación *O*P tendrá entonces la forma:

$$Op \mathrel{\widehat{=}} OpEstadoNoAfectado \land (OpYes \lor OpRechazo)$$

donde,

$$OpEstadoNoAfectado \mathrel{\widehat{=}} \Xi E_1 \land \ldots \land \Xi E_n$$
$$OpRechazo \mathrel{\widehat{=}} \Xi E_{n+1} \land \ldots \land \Xi E_m \land (OpNo \lor OpEr_1 \lor \ldots \lor OpEr_r)$$

Se entiende por E_i en $[1 \ldots n]$ un parte del estado que no es afectado por ningún caso de la operación. Y por E_j en $[n+1 \ldots m]$ un estado que es afectado cuando la ejecución de la operación tiene éxito y no cuando la ejecución se rechaza. Las partes de estado $E_1, \ldots, E_n$ son disjuntas con $E_{n+1}, \ldots, E_m$.

5.5.3 GetRead

Comenzaremos a definir ahora la primera y más importante operación de este trabajo. La operación *GetRead* entrega a cierto sujeto permiso de lectura a cierto objeto. Este tipo de acceso es el principal comprometedor de la seguridad del sistema, por estar la política orientada a la confidencialidad. Habrá que diseñar con cuidado las condiciones para la entrega de un permiso de lectura de manera que no genere un compromiso de seguridad.

Preservación de seguridad

Delinearemos ahora cuales son estas condiciones. Para esto recordemos que la política de seguridad consta de dos predicados: la propiedad de seguridad simple y la propiedad-★.

La propiedad de seguridad simple rige sobre aquellos sujetos que tienen permisos que permiten extracción de información del objeto verificando que ninguno de ellos posea una clase de seguridad que no domine la clase de seguridad de la información que está accediendo.

Esto significa que todo sujeto que invoca la operación *GetRead* deberá validar este predicado al cedérsele el permiso. Escrito en términos de lo ya especificado, lo anterior equivale a la siguiente expresión:

$$usrclass\ u\ dominates\ objclass\ o$$

donde u es el sujeto que desea el permiso de lectura y o es el objeto que éste desea acceder. Si el sujeto u que invoca al operación *GetRead* satisface este predicado entonces el permiso puede entregársele según la propiedad de seguridad simple.

Por otra parte, la propiedad-★ rige respecto de un permiso de lectura sobre aquellos objetos que el mismo sujeto está escribiendo, verificando que ninguno de ellos tenga una clase de seguridad que no domine la del objeto siendo leído.

Formalmente lo expresamos de la siguiente manera:

$$\forall\, wo : objSAtts\ (\theta ActualPermSet, u, \{Write, ReadWrite\}) \\ \bullet\ objclass\ wo\ dominates\ objclass\ o$$

donde wo son los objetos para los cuales u tiene acceso de escritura y o es el objeto para el cual el mismo requiere acceso de lectura.

Estas condiciones deberían ser suficientes para preservar la política de seguridad en *GetRead*, aunque la verificación formal de esto se verá en el próximo capítulo.

Caso válido de ejecución

Definiremos ahora el esquema *GetReadOkYes*. Éste representa el caso de la operación en el cual la misma actualiza satisfactoriamente el estado del módulo entregando el acceso requerido al sujeto que hizo el requerimiento. Daremos ahora su definición y luego explicaremos parte por parte el esquema.

GetReadOkYes

$$
\begin{array}{l}
AugmentingAccess[a := Read, u?/u, o?/o] \\
SecClasses \\
ACLS \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \\
(u?, Read) \in acl\ o? \\
usrclass\ u?\ dominates\ objclass\ o? \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Write, ReadWrite\}) \\
\quad \bullet\ objclass\ o\ dominates\ objclass\ o? \\
out! = YES
\end{array}
$$

Comenzaremos por la parte superior. Notar que hemos utilizado el esquema previamente definido *AugmentingAccess*. El mismo encapsula gran parte de la funcionalidad de *GetRead*, ya que representa una operación genérica que entrega un permiso de acceso. Lo hemos instanciado con *Read*, el permiso de lectura. También los parámetros u y o fueron renombrados a $u?$ y $o?$ respectivamente. Esto define las variables $u?$ y $o?$ como parámetros de entrada de la operación *GetReadOkYes*. La definición de variables como parámetros de entrada o de salida (? o !) nos provee propiedades extras para el manejo esquemas en Z. Entre otras, facilita la especificación operaciones en serie donde las subsecuentes toman como entrada la salida de las anteriores. Además de la instanciación del esquema *AugmentingAccess* hemos incluidos los esquemas que definen las partes discrecionaria y obligatoria del estado del módulo de control de acceso, respectivamente, *ACLS* y *SecClasses*. Necesitamos estos últimos para especificar los controles de seguridad.

Continuando por la parte inferior, la sección de invariantes, notamos cinco predicados, que corresponden a los cinco requerimientos necesarios para la ejecución de la operación. El primero especifica que tanto el objeto o como el sujeto u deben ser válidos. Seguido de esto, se requiere que el permiso de lectura asociado al sujeto u se encuentre en la ACL del objeto o. Luego se valida la propiedad de seguridad simple. Ya hemos analizado esta guarda al comenzar la especificación de *GetRead*, sólo se han renombrado las variables que representan al sujeto y objeto. Lo mismo pasa con el cuarto predicado, donde se valida la propiedad-★. Éste es el predicado ya analizado. Por último, indicaremos mediante la variable de salida *out* que no ha ocurrido un error al validar la ejecución de la operación y la misma puede ejecutarse normalmente.

Caso de rechazo

Seguiremos entonces con los casos de rechazo, caso que se da cuando las precondiciones de seguridad de la operación impiden que esta continúe su ejecución.

GetReadOkNo es el esquema que se aplica sobre el caso en el cual la operación

GetRead es invocada pero no se le es permitida su ejecución a causa de que compromete la seguridad del sistema. En este caso, alguno de los predicados asociados con la política obligatoria o la discrecionaria que no se satisface, pero el resto lo hace. La definición sigue:

GetReadOkNo

$$
\begin{array}{l}
SecClasses \\
ACLS \\
ActualPermSet \\
u? : UID \\
o? : OID \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \\
out! = NO \\
\neg\ ((u?, Read) \in acl\ o? \\
\quad \land\ usrclass\ u?\ dominates\ objclass\ o? \\
\quad \land\ (\exists\ o : objSAtts\ (\theta ActualPermSet, u?, \{Write, ReadWrite\}) \\
\qquad \bullet \neg\ objclass\ o\ dominates\ objclass\ o?)\ )
\end{array}
$$

El valor asociado a la variable de salida *out!* es *NO* y los predicados de control $o? \in oids$ y $u? \in uids$ deben ser válidos. Sin embargo no se cumple que todas las guardas de seguridad se satisfacen, la tercer sección del los invariantes expresa esto último.

Casos de error

Definiremos ahora los errores en invocación posibles de *GetRead*. El primer caso a cubrir es cuando $u?$ no referencia a un sujeto válido. Por esto el predicado $u? \in uids$ no vale.

Así, el esquema *GetReadEInvalidUser* se define de la siguiente manera:

$$GetReadEInvalidUser \mathrel{\widehat{=}} ErrorInvalidUser[function := fGetRead]$$

Siguiendo la misma idea, el error correspondiente a la invalidez de una referencia a un objeto se define de la siguiente manera:

$$GetReadEObjNotExist \mathrel{\widehat{=}} ErrorObjDoesNotExist\ [function := fGetRead]$$

GetRead completa

Ahora, de acuerdo con el estilo que mostramos para la definición de las operaciones, sólo queda definir las partes del estado afectadas por la operación, su complemento, y unir todas las partes para dar una especificación completa de la operación.

Si *GetRead* finaliza su ejecución sin fallar, entonces se actualizará la función *permSet* del esquema *ActualPermSet*. Ningún otro elemento del estado será afectado. Con lo cual la única parte afectada es *ActualPermSet*, el resto conforma el estado no afectado por esta operación:

$$\begin{aligned} GetReadNotAffected \mathrel{\widehat{=}} & \Xi UsrObj \land \\ & \Xi SecClasses \land \\ & \Xi ACLS \land \\ & \Xi MACAdmin \end{aligned}$$

finalmente, agrupamos los casos de rechazo y los de error:

$$\begin{aligned} GetReadReject \mathrel{\widehat{=}} \Xi ActualPermSet \land (& GetReadOkNo \lor \\ & GetReadEInvalidUser \lor \\ & GetReadEObjNotExist) \end{aligned}$$

y llegamos a la definición de *GetRead*:

$$\begin{aligned} GetRead \mathrel{\widehat{=}} GetReadNotAffected \land (& GetReadOkYes \lor \\ & GetReadReject) \end{aligned}$$

5.5.4 GetWrite

La operación *GetWrite* es análoga en muchos aspectos a la operación *GetRead* pero para el permiso de acceso de solo escritura (*Write*).

Una de las diferencias principales es que esta operación no requiere de verificación de la propiedad de seguridad simple, ya que esta última solo se aplica sobre los accesos que permiten extraer información.

Con respecto al resto de las guardas, se mantienen en analogía con las de *GetRead*.

GetWriteOkYes

$$\begin{aligned} & AugmentingAccess[u?/u, o?/o, a := Write] \\ & SecClasses \\ & ACLS \\ & out! : DECISION \\ \hline & o? \in oids \land u? \in uids \\ & (u?, Write) \in acl\ o? \\ & \forall o : objSAtts\ (\theta ActualPermSet, u?, \{Read, ReadWrite\}) \\ & \quad \bullet\ objclass\ o?\ dominates\ objclass\ o \\ & out! = YES \end{aligned}$$

No presentaremos aquí el resto de los esquemas que definen el esquema *GetWrite* ya que son muy similares a los que componen a *GetRead*[5]. Así, de manera

[5]Una versión completa de la especificación fue incluida en el apéndice B.

bastante directa, siguiendo las lineas de trabajo que ya utilizamos para *GetRead*, el esquema *GetWrite* queda definido como sigue:

$$GetWrite \mathrel{\widehat=} GetWriteNotAffected \land (GetWriteOkYes \lor GetWriteReject)$$

5.5.5 GetExecute

Siguiendo dentro de las operaciones que entregan permisos de acceso a objetos, definiremos ahora la operación *GetExecute*. Ésta entrega un permiso de ejecución sobre un objeto.

Debido a que esencialmente la ejecución de un objeto no compromete la seguridad a menos que la política discrecionaria diga explicitamente que no puede ser ejecutado por el sujeto en cuestión, y salvando lo que el programa ejecutado (si existe como tal) pueda llegar a hacer después, no es necesario poner guardas relacionadas con la política obligatoria. Posteriores accesos por parte del programa se asume que también serán comprobados utilizando el módulo de control de acceso. Así, tampoco es necesario macroexpandir el esquema *SecClasses*.

GetExecuteOkYes

$$AugmentingAccess[u?/u, o?/o, a := Execute]$$
$$ACLS$$
$$out! : DECISION$$

$$o? \in oids \land u? \in uids$$
$$out! = YES$$
$$(u?, Execute) \in acl\ o?$$

Nuevamente, y como en las operaciones anteriores, en parte debido a nuestro marco de trabajo y en parte a la similitud de estas nuevas operaciones con las anteriores, la definición del resto de los esquemas que definen al esquema *GetExecute* resulta directa y no será incluida aquí.

5.5.6 GetReadWrite

La última de las operaciones que entregan un permiso de acceso a un sujeto, *GetReadWrite*, requiere guardas más complejas que las anteriores. Esto se debe a la funcionalidad de alguna manera aumentada del permiso de acceso *ReadWrite*.

Las guardas correspondientes a la política ACL siguen siendo análogas, lo mismo que la asociada a la propiedad de seguridad simple. Pero la guarda que protege la propiedad-★ es más restrictiva, e incluye una conjunción de las guardas vista en los casos *GetRead* y *GetWrite*.

GetReadWriteOkYes

$$
\begin{array}{l}
AugmentingAccess[u?/u, o?/o, a := ReadWrite] \\
SecClasses \\
ACLS \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \\
(u?, ReadWrite) \in acl\ o? \\
usrclass\ u?\ dominates\ objclass\ o? \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Read\}) \bullet \\
\quad objclass\ o?\ dominates\ objclass\ o \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Write\}) \bullet \\
\quad objclass\ o\ dominates\ objclass\ o? \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{ReadWrite\}) \bullet \\
\quad objclass\ o?\ matches\ objclass\ o \\
out! = YES
\end{array}
$$

Las tres guardas

$$
\begin{array}{l}
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Read\}) \bullet \\
\quad objclass\ o?\ dominates\ objclass\ o \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Write\}) \bullet \\
\quad objclass\ o\ dominates\ objclass\ o? \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{ReadWrite\}) \bullet \\
\quad objclass\ o?\ matches\ objclass\ o
\end{array}
$$

equivalen a estas dos

$$
\begin{array}{l}
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Read, ReadWrite\}) \bullet \\
\quad objclass\ o?\ dominates\ objclass\ o \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Write, ReadWrite\}) \bullet \\
\quad objclass\ o\ dominates\ objclass\ o?
\end{array}
$$

ya que vale o_1 *matches* $o_2 \Leftrightarrow o_1$ *dominates* $o_2 \land o_2$ *dominates* o_1 (§ 5.4.1).

En la última disposición de la guarda se ve claramente que es la conjunción de las guardas asociadas a las propiedad-★ de las operaciones *GetRead* y *GetWrite*.

5.5.7 Release

Release es la operación que abandona un permiso de acceso. Siempre que un sujeto pide un permiso de acceso determinado a través de alguna de las operaciones anteriores, debe devolverlo cuando termina de acceder al objeto asociado, y lo hace a través de *Release*.

Esta operación no es una operación que comprometa la seguridad, por eso no veremos ninguna guarda como las anteriores asociada con esta operación. Por supuesto, esto no la excluye de la prueba formal de que el sistema no entra en un compromiso de seguridad al ejecutarla. Sí encontraremos los casos de errores de ejecución ya manifestados.

Aquí haremos uso de la función *dimb* antes mencionada (como análoga de *augb*), y en seguida la definiremos.

ReleaseOk

$$
\begin{array}{l}
\Delta ActualPermSet \\
u? : UID \\
o? : OID \\
a? : PERMISSION \\
out! : DECISION \\
\hline
o? \in oids \\
u? \in uids \\
permSet' = dimb\ (permSet, (u?, o?), a?) \\
out! = YES
\end{array}
$$

Como análoga de *augb*, *dimb* permite expresar fácilmente que queremos eliminar un permiso de acceso existente del estado del módulo de control de acceso. La aplicación *dimb* $(permSet, (u, o), a)$ representa la función que equivale a *permSet* salvo en (u, o), cuya imagen excluye el permiso de acceso a. La definiremos de la siguiente manera:

$$
\begin{array}{l}
dimb : fUO\_PERMS \times UOp \times PERMISSION \rightarrow fUO\_PERMS \\
\hline
\mathrm{dom}\ dimb = \{f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid \\
\qquad up \in \mathrm{dom}\ f\} \\
\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid up \in \mathrm{dom}\ f \\
\quad \bullet\ dimb\ (f, up, na) = f \oplus \{(up \mapsto fup \setminus \{na\})\}
\end{array}
$$

Notar que si bien el uso más indicado de la función *dimb* es el de eliminar un permiso de acceso existente, la existencia de dicho permiso de acceso no es una condición necesaria para su utilización, pudiendo cubrir el caso en el cual el permiso de acceso no existe para el par referenciado en la misma expresión.

5.5.8 Give y Rescind

Las operaciones *Give* y *Rescind* gestionan la parte discrecionaria de la política de seguridad. Ambas implican exclusivamente actualizar las ACLs asociadas al objeto a las que se aplican.

Utilizando *Give*, un sujeto puede entregar un derecho de acceso a otro sujeto, si es que dispone del mismo y si es que, además, tiene derecho de control sobre el objeto (*Control*).

GiveOkYes

$$
\begin{array}{l}
\Delta ACLS \\
u? : UID \\
o? : OID \\
a? : PERMISSION \\
target? : UID \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \land target? \in uids \\
out! = YES \\
(u?, a?) \in acl\ o? \\
(u?, Control) \in acl\ o? \\
acl' = acl \oplus \{(o? \mapsto acl\ o? \cup \{(target?, a?)\})\}
\end{array}
$$

En contrapartida a *Give*, *Rescind* es la operación que quita un derecho de acceso. Una vez más, elegimos que el sujeto que invoca la operación, tenga los derechos de acceso correspondientes, es decir, el que desea quitar y *Control*. Así, su definición resulta muy similar a la de *Give*:

RescindOkYes

$$
\begin{array}{l}
\Delta ACLS \\
u? : UID \\
o? : OID \\
a? : PERMISSION \\
target? : UID \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \land target? \in uids \\
out! = YES \\
(u?, a?) \in acl\ o? \\
(u?, Control) \in acl\ o? \\
acl' = acl \oplus \{(o? \mapsto acl\ o? \setminus \{(target?, a?)\})\}
\end{array}
$$

5.5.9 ChangeObjClass

La operación *ChangeObjClass* permite cambiar la clase de seguridad de un objeto. Sólo puede utilizarse cuando nadie está accediendolo, y sólo si el usuario es

un administrador de la política de seguridad obligatoria. Por esto, la siguente operación incluye el esquema ya definido *MACAdmin*:

ChangeObjClassOkYes

$$
\begin{array}{l}
ActualPermSet \\
MACAdmin \\
\Delta SecClasses \\
u? : UID \\
o? : OID \\
secClass? : SecClass \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \land u? \in macAdmins \\
\forall u : uids \bullet permSet\ (u, o?) = \emptyset \\
out! = YES \\
objclass' = objclass \oplus \{(o? \mapsto secClass?)\} \\
usrclass' = usrclass
\end{array}
$$

5.5.10 CreateObject y DeleteObject

Las últimas dos operaciones que definiremos conciernen la creación y eliminación de objetos en el sistema.

Primero especificaresmos la operación *CreateObject*, que permite crear un objeto en el sistema. Esto implica ampliar el conjunto *oids*, y con él, el dominio de las funciones que se aplican sobre este conjunto. En ambos casos, las funciones *acl*, *objclass* y *permSet* necesitarán actualizarse en el transcurso de la operación para que el estado del módulo de control de acceso preserve su consistencia.

$$InitOSecClass \mathrel{\widehat=} BottomSecClass$$

CreateObjectOkYes

$$
\begin{array}{l}
\Delta ACLS \\
\Delta ActualPermSet \\
\Delta SecClasses \\
MACAdmin \\
u? : UID \\
o? : OID \\
out! : DECISION \\
\hline
u? \in uids \land o? \notin oids \land u? \in macAdmins \\
out! = YES \\
oids' = oids \cup \{o?\} \land uids' = uids \\
acl' = acl \oplus \{(o? \mapsto \emptyset)\} \\
objclass' = objclass \oplus \{(o? \mapsto InitOSecClass)\} \\
usrclass' = usrclass \\
permSet' = permSet \oplus \{u : uids \bullet ((u, o?) \mapsto \emptyset)\}
\end{array}
$$

Arbitrariamente se ha definido que la clase de los nuevos objetos es la menor clase posible del sistema, dado que el creador puede darle la clase que quiera posteriormente con la operación *ChangeObjClass*.

Similar a *CreateObject*, *DeleteObject*, en contrapartida, tiene por fin eliminar un objeto del sistema.

DeleteObjectOkYes

$\Delta ACLS$
$\Delta ActualPermSet$
$\Delta SecClasses$
$MACAdmin$
$u? : UID$
$o? : OID$
$out! : DECISION$

$u? \in uids \land o? \in oids \land u? \in macAdmins$
$oids' = oids \setminus \{o?\} \land uids' = uids$
$permSet' = \{u : uids \bullet (u, o?)\} \lhd permSet$
$acl' = \{o?\} \lhd acl$
$usrclass' = usrclass$
$objclass' = \{o?\} \lhd objclass$
$out! = YES$

5.6 Preservación de la Política de Seguridad

Para imponer la política de seguridad, el módulo de control de acceso necesita certificar que el sistema no entra en un estado que compromete la seguridad. Basándonos en el concepto de monitor de referencias, asumimos ahora que el módulo de control de acceso es la única interfaz posible para acceder a los datos. Sólo las operaciones podrían introducir un compromiso de seguridad. Una vez que podemos asegurar que esto ocurre sólo a través de las operaciones, entonces queda en estas últimas mantener vigente la política.

Definiremos entonces los predicados que deberán satisfacer las operaciones a fin de preservar la política de seguridad.

5.6.1 Propiedad de Seguridad Simple

Una de las propiedades que queremos preservar en todos los estados es la propiedad de seguridad simple, definida en la sección 5.4.3 en forma de un esquema denominado *SimpleSecureState*.

Una de las herramientas que utilizamos en Z para definir operaciones que actualizan esquemas es Δ. Recordemos que ΔA macroexpande los esquemas A y A', correspondientes al esquema A antes y después de la operación. Recurri-

remos nuevamente a esta técnica aunque esta vez escribiremos los esquemas primados directamente, no a través del operador Δ. Así, podemos decir que una operación que preserva seguridad debe satisfacer:

$$(SimpleSecureState \Rightarrow SimpleSecureState')$$

Completando un poco la expresión con las partes que intervienen en el predicado *SimpleSecureState* para aclarar el contexto de la implicación, la definición del predicado buscado, *PreservesSimpleSecurity*, sigue:

$$\begin{aligned} PreservesSimpleSecurity \mathrel{\widehat=} & \Delta ActualPermSet \wedge \\ & \Delta SecClasses \wedge \\ & (SimpleSecureState \Rightarrow SimpleSecureState') \end{aligned}$$

Así, para afirmar que una operación *OP* preserva la propiedad de seguridad simple, expresamos:

$$OP \Rightarrow PreservesSimpleSecurity$$

Una versión opcional, utilizando directamente *SimpleSecureState*, podría ser:

$$SimpleSecureState \wedge OP \Rightarrow SimpleSecureState'$$

5.6.2 Propiedad-★

La preservación de la propiedad-★ se define de manera equivalente. El predicado que le corresponde, definido en la sección 5.4.4, es *StarPropertyState*. Luego del mismo razonamiento que en la sección anterior, expresamos el predicado que debe satisfacer una operación para preservarla, *PreservesStarProperty*, de la siguiente manera:

$$\begin{aligned} PreservesStarProperty \mathrel{\widehat=} & \Delta ActualPermSet \wedge \\ & \Delta SecClasses \wedge \\ & (StarPropertyState \Rightarrow \\ & StarPropertyState') \end{aligned}$$

5.6.3 Conjunción de las propiedades

Terminando la sección, concluimos que las operaciones, a fin de preservar la política de seguridad MLS, deberán preservar a la vez los predicados anteriores. Para una operación *O*P, si esta satisface el siguiente predicado, entonces preserva tanto la propiedad de seguridad simple como la propiedad-★:

$$OP \Rightarrow PreservesSimpleSecurity \wedge PreservesStarProperty$$

Capítulo 6

Verificación

Esta segunda etapa de nuestro trabajo consiste en probar que nuestra especificación es consistente con nuestros objetivos; es decir, cumple con las propiedades deseadas.

Debido a que el trabajo necesario para probar las propiedades en forma formal, y en especial mediante un asistente de pruebas requiere de mucho esfuerzo y tiempo, se incluye en este trabajo sólo una del total pruebas. La prueba elegida es la que demuestra consistencia de la operación *GetRead* respecto de la política de seguridad MLS. La elección se basa en que la idea principal detrás de la política MLS de BLP es la protección de confidencialidad, lo que convierte a la *GetRead* la principal causante de compromisos de seguridad.

El proceso para probar esto equivale al de cualquiera de las pruebas de consistencia de las operaçiones. Con mayores o menores esfuerzos dependiendo de los riesgos de la operación.

Para probar que la operación *GetRead* preserva la seguridad especificaremos un marco de trabajo con un poco más de desarrollo del estrictamente necesario. Luego de esto, la prueba del teorema *GetReadIsSound* es nuestro objetivo final.

La pruebas desarrolladas para este teorema, que incluyen una gran cantidad de lemas auxiliares, sin duda relajan el esfuerzo necesario para futuras pruebas.

6.1 Mecanismo de prueba de Z/EVES

Z/EVES 2.1 provee un mecanismo de pruebas que permite aplicar tanto estrategias automáticas como instrucciones que implementan reglas deductivas elementales. El mecanismo se basa en traducir la especificación a cálculo de predicados de primer orden sin tipo, razonar en esta lógica, y luego volver a Z. El tipado se expresa utilizando pertenencia de conjuntos.

Cada prueba en Z/EVES se lleva a cabo bajo una *teoría* particular. Esta *teoría*, diferente para cada propiedad, está constituida por la base que provee el asistente de pruebas más los axiomas, lemas y teoremas que fueron definidos antes del que está siendo probado.

Z/EVES utiliza el siguiente mecanismo para probar teoremas. Un teorema está constituido por un predicado *objetivo* que se debe probar. Cada paso de una prueba *transforma* el objetivo en otro equivalente[1] sucesivamente hasta que es transformado a *true*.

Una prueba se reduce entonces a una lista de comandos, cada uno transformando el objetivo hasta llegar al último paso de la prueba que lo transforma a *true*.

6.2 Lemas de BLP

Algunas de las propiedades que Bell y LaPadula ya mostraron sirven de guía para las que desarrollaremos. Esto se da especialmente en los lemas auxiliares, ya que son muchas veces independientes del la forma en la que se especificó el módulo y por lo tanto resultan más reusables.

Debido a que la forma en que se probarán los teoremas es radicalmente diferente en este trabajo, no es posible reusar muchos de los razonamientos efectuados en las pruebas de BLP. Varios enunciados de lemas son análogos a los de BLP, pero las pruebas fueron totalmente reconstruidas.

Indicaremos que estamos basandonos en un lema que ya aparecía en BLP cada vez que esto ocurra.

[1] Salvo en ciertas excepciones [13]

6.3 Estrategia para la prueba de consistencia

Al final del capítulo 5 explicamos la forma de enunciar que cierta operación preserva las propiedades de seguridad. Enunciaremos ahora, en base ese patrón, el teorema de consistencia para *GetRead*:

Theorem GetReadIsSound
$\forall$ *GetRead* • *PreservesSimpleSecurity* $\wedge$ *PreservesStarProperty*

Debido a la forma de los teoremas en Z/EVES y al mecanismo de pruebas, hemos alterado ligeramente la sintaxis. Esta nueva forma es equivalente a la presentada en el capítulo anterior.

6.3.1 Factorización de la prueba

Debido a la forma composicional que dimos a la especificación, y gracias a algunas generalizaciones y factorizaciones de conceptos, podremos subdividir la prueba de *GetRead* en varios teoremas más simples. Esto nos permitirá razonar más focalizadamente durante las pruebas, e incluso nos permite utilizar más a menudo las estrategias de prueba automátizadas de Z/EVES 2.1.

Buscaremos dividir la prueba en casos analizando primero el consecuente y luego el antecedente del teorema.

6.3.2 Casos derivados del consecuente

Debido a que la conclusión del teorema es una conjunción de dos predicados importantes, la primera línea de trabajo que adoptaremos para probar el teorema será dividir las pruebas en dos casos, uno para cada componente de la conjunción.

Cambiaremos la implicación:

GetRead $\Rightarrow$ *PreservesSimpleSecurity* $\wedge$ *PreservesStarProperty*

por dos implicaciones:

GetRead $\Rightarrow$ *PreservesSimpleSecurity*
GetRead $\Rightarrow$ *PreservesStarProperty*

Ahora en vez de una prueba tenemos dos independientes, una para cada predicado, pero veremos que estas todavía tienen mucho en común. Aunque con esto estamos descartando posibles razonamientos en común entre ambas pruebas, compensaremos esta estrategia de separación con el desarrollo de teoremas auxiliares que podremos reutilizar en ambas ramas.

6.3.3 Casos derivados del antecedente

El predicado *GetRead* está definido en una serie de niveles de abstracción que permiten modularizar la prueba desde el punto de vista recíproco al recién utilizado.

Si en la definición de *GetRead* (§ 5.5.3) aplicamos distributividad de $\wedge$ con respecto a $\vee$, tenemos

$$(GetReadNotAffected \wedge GetReadOkYes) \vee \\ (GetReadNotAffected \wedge GetReadReject)$$

Por un lado, los esquemas *GetReadReject* y *GetReadNotAffected* definen la rama de la operación que no modifica el estado del módulo. Esto debería verificar sin problemas que la seguridad se preserva, ya que el estado no cambia y por lo tanto no puede introducirse un compromiso de seguridad. En el otro caso, la operación queda definida por *GetReadOkYes* y actualiza el estado ya que las guardas se cumplen. Es claro que la prueba es mucho más compleja en este caso comparada con el de los casos de rechazo. En los casos de rechazo, el teorema queda casi probado por sus hipótesis, y el otro requiere una línea de razonamiento más compleja.

Con este plan, haremos la siguiente división de casos:

$A \cdot (GetReadNotAffected \wedge GetReadOkYes) \Rightarrow PreservesSimpleSecurity$

$B \cdot (GetReadNotAffected \wedge GetReadOkYes) \Rightarrow PreservesStarProperty$

$C \cdot (GetReadNotAffected \wedge GetReadReject) \Rightarrow PreservesSimpleSecurity$

$D \cdot (GetReadNotAffected \wedge GetReadReject) \Rightarrow PreservesStarProperty$

6.4 Teoremas auxiliares para casos de rechazo

En esta sección desarrollaremos dos teoremas que nos ayudarán a probar los casos del teorema *GetReadIsSound* en los cuales la operación no puede tener lugar y es rechazada, es decir, los casos *C* y *D*.

6.4.1 Propiedad de Seguridad Simple

Enunciaremos el primer teorema que servirá a probar *GetReadIsSound*:

Theorem SimpleSecurityNotAffected
$\Xi ActualPermSet \wedge \Xi SecClasses \Rightarrow PreservesSimpleSecurity$

El teorema *SimpleSecurityNotAffected* enuncia que si el estado no a cambiado, o al menos no han cambiado ni *SecClasses* ni *ActualPermSet*, entonces la propiedad de seguridad simple se preserva. La demostración de este teorema, que sigue a continuación, resulta casi trivial bajo Z/EVES.

proof)
invoke *PreservesSimpleSecurity*
invoke Ξ*ActualPermSet*
invoke Ξ*SecClasses*
invoke Δ*ActualPermSet*
invoke Δ*SecClasses*
simplify
■

La primera instrucción utilizada en la prueba, y de hecho la utilizada en los cinco primeros pasos, es la instrucción *invoke*. La sentencia

invoke *PreservesSimpleSecurity*

expande el esquema *PreservesSimpleSecurity* según su definición. Como hemos dicho en la sección de introducción, Z/EVES presenta el predicado a probar y las sentencias ejecutadas aplican transformaciones que pretenden llevarlo a *true* y así probar su validez. En el caso de esta prueba, partimos del siguiente enunciado:

$$\begin{array}{l} \quad \Xi ActualPermSet \\ \land\ \Xi SecClasses \\ \Rightarrow (PreservesSimpleSecurity \Leftrightarrow true) \end{array}$$

que luego de la sentencia *invoke PreservesSimpleSecurity* (§ 5.6.1) es transformado a

$$\begin{array}{l} \quad \Xi ActualPermSet \\ \land\ \Xi SecClasses \\ \Rightarrow \Delta ActualPermSet \land \\ \quad \Delta SecClasses \land \\ \quad (SimpleSecureState \Rightarrow SimpleSecureState') \end{array}$$

Luego, mediante la instrucción *invoke* Ξ*ActualPermSet*, y por la definición del operador Ξ, el objetivo se transforma de la siguiente manera:

$$\begin{array}{l} \quad ActualPermSet \land ActualPermSet' \\ \land\ \theta ActualPermSet = \theta ActualPermSet' \\ \land\ \Xi SecClasses \\ \Rightarrow \Delta ActualPermSet \land \\ \quad \Delta SecClasses \land \\ \quad (SimpleSecureState \Rightarrow SimpleSecureState') \end{array}$$

La cuarta instrucción, *invoke* $\Xi ActualPermSet$, deja como objetivo la siguiente expresión:

$$\begin{array}{l} \quad ActualPermSet \wedge ActualPermSet' \\ \wedge\ \theta ActualPermSet = \theta ActualPermSet' \\ \wedge\ SecClasses \wedge SecClasses' \\ \wedge\ \theta SecClasses = \theta SecClasses' \\ \Rightarrow ActualPermSet \wedge ActualPermSet' \\ \quad \wedge\ SecClasses \wedge SecClasses' \\ \quad \wedge\ (SimpleSecureState \Rightarrow SimpleSecureState') \end{array}$$

Por último, la operación *simplify* concluye la prueba.

true

El comando *simplify* en un comando de automatización. Los comandos con estrategias automatizadas aplican reglas de deducción en busca de una transformación que haga verdadero al objetivo o que lo acerque un poco en esta dirección. Hay varios de estos comandos, cada uno con su propia estrategia de prueba. En particular, *simplify* realiza razonamiento sobre predicados de igualdad y sobre números enteros, hace análisis proposicional y chequeo de tautologías, además de aplicar *forward rules* y *assumption rules* cuando es posible [13].

Así, los predicados *ActualPermSet*, *ActualPermSet'*, *SecClasses*, y *SecClasses'* se cancelan del consecuente por ser parte del antecedente. Además, dado que *SimpleSecureState* sólo menciona *uids*, *oids*, *permSet* y la macroexpansión de *SecClasses*, y que $\theta ActualPermSet = \theta ActualPermSet'$ y $\theta SecClasses = \theta SecClasses'$, todas las variables de *SimpleSecureState* equivalen a sus primadas. El comando *simplify* hace uso de algo de razonamiendo sobre predicados de igualdad y determina que *SimpleSecureState* equivale a *SimpleSecureState'*. Por lo tanto vale $SimpleSecureState \Rightarrow SimpleSecureState'$ y el objetivo se transforma en *true*.

De esta manera concluye la demostración de *SimpleSecurityNotAffected*, probando así que $\Xi ActualPermSet \wedge \Xi SecClasses \Rightarrow PreservesSimpleSecurity$.

Usaremos luego este teorema para probar el caso *C*.

6.4.2 Propiedad-★

Enunciaremos ahora el teorema análogo a *SimpleSecurityNotAffected* esta vez para la propiedad-★: *StarPropertyNotAffected*.

Corresponde a este teorema enunciar que si nada ha cambiado en las etiquetas que clasifican a los objetos y sujetos del sistema ($\Xi SecClasses$), y nada ha

cambiado en los permisos de acceso ($\Xi ActualPermSet$), entonces debe valer *PreservesStarProperty*:

Theorem StarPropertyNotAffected
$\Xi ActualPermSet \land \Xi SecClasses \Rightarrow PreservesStarProperty$

Como en la prueba anterior, primero tienen lugar una serie de instrucciones *invoke*, las siguientes:

invoke *PreservesStarProperty*
invoke $\Delta ActualPermSet$
invoke $\Delta SecClasses$
invoke *StarPropertyState*

Al ejecutar estas sentencias, el objetivo queda transformado en la siguiente expresión:

$$
\begin{array}{l}
\quad \Xi ActualPermSet \land \Xi SecClasses \\
\Rightarrow ActualPermSet \\
\quad \land\ ActualPermSet' \\
\quad \land\ SecClasses \\
\quad \land\ SecClasses' \\
\quad \land\ (ActualPermSet \\
\qquad \land\ SecClasses \\
\qquad \land\ (\forall u : UID;\ ow : OID;\ or : OID \\
\qquad\quad \mid u \in uids \\
\qquad\qquad \land\ ow \in objSAtts\ (\theta ActualPermSet, u, \\
\qquad\qquad\qquad\qquad (\{Write\} \cup \{ReadWrite\})) \\
\qquad\qquad \land\ or \in objSAtts\ (\theta ActualPermSet, u, \\
\qquad\qquad\qquad\qquad (\{Read\} \cup \{ReadWrite\})) \\
\qquad\quad \bullet\ objclass\ ow\ dominates\ objclass\ or) \\
\qquad ActualPermSet' \\
\qquad \land\ SecClasses' \\
\qquad \land\ (\forall u\_0 : UID;\ ow\_0 : OID;\ or\_0 : OID \\
\qquad\quad \mid u\_0 \in uids' \\
\qquad\qquad \land\ ow\_0 \in objSAtts\ (\theta ActualPermSet', u\_0, \\
\qquad\qquad\qquad\qquad (\{Write\} \cup \{ReadWrite\})) \\
\qquad\qquad \land\ or\_0 \in objSAtts\ (\theta ActualPermSet', u\_0, \\
\qquad\qquad\qquad\qquad (\{Read\} \cup \{ReadWrite\})) \\
\qquad\quad \bullet\ objclass'\ ow\_0\ dominates\ objclass'\ or\_0))
\end{array}
$$

Notar que la complejidad del objetivo comienza a crecer excesivamente luego de estos pasos. Los objetivos pueden crecer exponencialmente en tamaño si se aplican solo reglas que lo agrandan como *invoke*. La regla *invoke* u otras reglas que hacen crecer en tamaño del objetivo expandiendo las componentes a fin de encontrar nuevos elementos para razonar son muy útiles, pero necesitaremos aplicar simplificaciones paulatinamente para poder razonar más claramente y para no sobrecargar a Z/EVES innecesariamente.

Ahora, con los comandos *simplify* y *prenex*, simplificamos algunos de los predicados expandidos y organizamos en forma prenexa el objetivo. La forma prenexa lleva los cuantificadores lo más afuera posible de la expresión, permitiendo luego una mejor manipulación de las variables. Los cuantificadores desaparecen del objetivo cuando lo engloban completamente.

simplify
prenex

Esto deja la expresión en una forma más clara y así podemos planear la siguiente movida más facilmente:

$$
\begin{array}{l}
\quad \Xi ActualPermSet \land \Xi SecClasses \\
\land\ (\forall u\_\_0 : UID;\ ow\_\_0 : OID;\ or\_\_0 : OID \\
\qquad \mid u\_\_0 \in uids \\
\qquad\quad \land\ ow\_\_0 \in objSAtts\ (\theta ActualPermSet, u\_\_0, \\
\qquad\qquad\qquad\qquad (\{Write\} \cup \{ReadWrite\})) \\
\qquad\quad \land\ or\_\_0 \in objSAtts\ (\theta ActualPermSet, u\_\_0, \\
\qquad\qquad\qquad\qquad (\{Read\} \cup \{ReadWrite\})) \\
\qquad \bullet\ objclass\ ow\_\_0\ dominates\ objclass\ or\_\_0) \\
\land\ u \in UID \land ow \in OID \land or \in OID \land u \in uids' \\
\land\ ow \in objSAtts\ (\theta ActualPermSet', u, \\
\qquad\qquad\qquad (\{Write\} \cup \{ReadWrite\})) \\
\land\ or \in objSAtts\ (\theta ActualPermSet', u, \\
\qquad\qquad\qquad (\{Read\} \cup \{ReadWrite\})) \\
\Rightarrow objclass'\ ow\ dominates\ objclass'\ or
\end{array}
$$

El cuantificador que aparece en el antecedente es el asociado con la propiedad-★ en el estado anterior a la operación. Y lo que tenemos que probar la propiedad vale sobre el estado destino.

Como $uids = uids'$, entonces $u \in uids$. Además, como $\theta ActualPermSet$ equivale a su versión primada, es trivial que $or \in objSAtts\ (\theta ActualPermSet, \ldots)$ y también que $ow \in objSAtts\ (\theta ActualPermSet, \ldots)$. Así, podremos instanciar $u\_\_0$ con u, $ow\_\_0$ con ow y $or\_\_0$ con or, y por lo anterior, valdrá $objclass\ ow\ dominates\ objclass\ or$.

Uno debe percatarse para estos casos de que los nombres de variables no persisten a lo largo de las pruebas en Z/EVES. Una variable con cierto nombre en cierto paso puede cambiar de nombre en el siguiente. Esto se da en las variables que necesitan ser renombradas por una colisión de nombres. Además, estas operaciones de renombre no se almacenan, son ejecutadas cada vez que se quiere mostrar el predicado. Si luego de haber probado un teorema se cambia algo de la estructura del predicado, por más leve que sea el cambio, los nombres generados pueden ser diferentes y la prueba dejaría de servir. Este es el caso de ow con $ow\_\_0$ y de or con $or\_\_0$, que luego *simplify* y *prenex* intercambian los nombres.

Instanciaremos ahora u_0, y luego de algo de reacomodación, haremos lo mismo con ow_0 y or_0:

instantiate $u\_0 == u$
rearrange
simplify
instantiate $ow\_1 == ow, or\_1 == or$
simplify

Y una vez hecho esto corremos otro de los mecanismos de automatización de Z/EVES, esta vez el de más alto nivel y con mayor capacidad de deducción.

prove by reduce
∎

Y el teorema *StarPropertyNotAffected* queda probado[2]. Este teorema será suficiente para probar el caso *D* del teorema *GetReadIsSound*.

6.5 Teoremas auxiliares para los casos válidos

Ahora pasaremos a desarrollar teoremas auxiliares para los casos más importantes de la operación *GetRead*, los casos *A* y *B*. En estos la operación *GetRead* tiene éxito y por lo tanto entrega el permiso de acceso requerido. Por supuesto, corresponde probar que preserva tanto la propiedad de seguridad simple como la propiedad-★:

$$A \cdot (GetReadNotAffected \land GetReadOkYes) \Rightarrow PreservesSimpleSecurity$$

$$B \cdot (GetReadNotAffected \land GetReadOkYes) \Rightarrow PreservesStarProperty$$

El caso en el cual la ejecución de la operación *GetRead* es permitida actualiza la función *permSet* del esquema *ActualPermSet*.

Recordemos que *GetRead* se define en base a *AugmentingAccess*, que define el comportamiento general de una operación que entrega un nuevo permiso de acceso. *AugmentingAccess* utiliza para esto la función *augb*. Otra función que se utiliza en el esquema *GetReadOkYes* es *objSAtts*, en la precondición referente a la propiedad-★. Esta última es una mera ayuda sintática.

Sin teoremas que resuman sus propiedades, las pruebas que vienen a continuación se verían contaminadas con razonamiento sobre las definición de estas dos funciones. Probablemente en numerosas ocaciones habría que probar las mismas

[2]Todas las pruebas completas de esta especificación pueden encontrarse en el apéndice C.

cosas. Esto entorpecería las pruebas sacándonos del foco principal y también complicaría los mecanismos automatizados de Z/EVES.

Para que las pruebas subsiguientes no se vuelvan demasiado largas y llenas de detalles fuera de la línea principal de demostración, enunciaremos cinco teoremas (o lemas) referentes a *augb* y tres referentes a *objSAtts*. En particular, definiremos para *objSAtts* teoremas que hablan sobre su interacción con la función *augb*. Así podremos evitar tener que expandir sus respectivas definiciones en cada prueba en la que se vean involucradas.

6.5.1 Teoremas para *augb*

Los cinco teoremas referentes a *augb* siguen:

Theorem rule augb_domFEqu
$$\forall g : fUO\_PERMS; f : fUO\_PERMS; up : UOp; na : PERMISSION \mid up \in \mathrm{dom}\, f \land g = augb\ (f, up, na) \bullet \mathrm{dom}\, g = \mathrm{dom}\, f$$

Theorem rule augb_aGrow
$$\forall g : fUO\_PERMS; f : fUO\_PERMS; up : UOp; na : PERMISSION; op : UOp \mid up \in \mathrm{dom}\, f \land g = augb\ (f, up, na) \land op \in \mathrm{dom}\, f \bullet f\ op \subseteq g\ op$$

Theorem rule augb_naEqu
$$\forall g : fUO\_PERMS; f : fUO\_PERMS; up : UOp; na : PERMISSION \mid up \in \mathrm{dom}\, f \land g = augb\ (f, up, na) \bullet \forall p : \mathrm{dom}\, f \mid p \neq up \bullet g\ p = f\ p$$

Theorem rule augb_aNoDim
$$\forall g : fUO\_PERMS; f : fUO\_PERMS; up : UOp; na : PERMISSION \mid up \in \mathrm{dom}\, f \land g = augb\ (f, up, na) \bullet \forall p : \mathrm{dom}\, f \bullet \forall a : g\ p \mid na \neq a \bullet a \in f\ p$$

Theorem rule augb_naNoDim
$$\forall g : fUO\_PERMS; f : fUO\_PERMS; up : UOp; na : PERMISSION \mid up \in \mathrm{dom}\, f \land g = augb\ (f, up, na) \bullet \forall p : \mathrm{dom}\, f \bullet \forall a : g\ p \mid (up, na) \neq (p, a) \bullet a \in f\ p$$

El teorema *augb_domFEqu* marca que luego de una actualización de una función f utilizando *augb* el dominio de la función no es alterado. El segundo,

augb_aGrow, predica sobre la nueva imagen de la función, y certifica que el conjunto de atributos del elemento actualizado incluye al original, es decir, crece o se mantiene igual. El tercero, *augb_naEqu*, indica que para el resto de los elementos del dominio de la función su imagen no cambia. El cuarto dice que si un atributo aparece en la nueva función para cierto objeto y éste no es del mismo tipo que el atributo agregado entonces seguro este ya estaba en la función antes de la actualización. Por último, *augb_naNoDim*, similar al anterior pero más complicado, dice que todos los atributos asociados a sujetos en la nueva función ya estaban en la función original, excepto en el caso del atributo agregado y el sujeto modificado.

Cabe mencionar que hay varios teoremas más que han sido desarrollados pero no mostraremos. Los mismos no resultan de mayor interés salvo para pequeñas ayudas puramente sintácticas, como por ejemplo algunas reglas de reescritura. Estos están incluidos en el apéndice C. Las pruebas de los teoremas recién enumerados están incluidas en el mismo apéndice. Debido a que son largas y la mayoría puramente técnicas, no se profundiza más sobre éstas y por lo tanto no serán incluidas en este capítulo.

6.5.2 Teoremas para *objSAtts* en su interacción con *augb*

Ahora enunciaremos los teoremas que hablan de la interacción entre *objSAtts* y *augb*:

Theorem objSAttsAug_aGrow
$$AugmentingAccess \land \Xi UsrObj \Rightarrow$$
$$(\forall ou : UID;\ pa : \mathbb{P}\, PERMISSION \mid ou \in uids \bullet$$
$$(\forall x : objSAtts\ (\theta ActualPermSet, ou, pa) \bullet$$
$$x \in objSAtts\ (\theta ActualPermSet', ou, pa)\ )\ )$$

Theorem objSAttsAug_na
$$AugmentingAccess \land \Xi UsrObj \Rightarrow$$
$$(\forall ou : UID;\ pa : \mathbb{P}\, PERMISSION \mid$$
$$ou \in uids \land (u \neq ou \lor a \notin pa) \bullet$$
$$(\forall x : objSAtts(\theta ActualPermSet', ou, pa) \bullet$$
$$x \in objSAtts(\theta ActualPermSet, ou, pa)\ )\ )$$

Theorem objSAttsAug_aDef
$$AugmentingAccess \land \Xi UsrObj \Rightarrow$$
$$(\forall pa : \mathbb{P}\, PERMISSION \mid a \in pa \bullet$$
$$(\forall x : objSAtts\ (\theta ActualPermSet', u, pa) \bullet$$
$$x \in objSAtts\ (\theta ActualPermSet, u, pa) \cup \{o\}\ )\ )$$

Notar que, además de describir propiedades de *objSAtts* en lo que respecta a las interacciones con *augb*, los teoremas se centran exclusivamente en el uso del

esquema *AugmentingAccess*. Todos ellos hablan de qué ocurre con la función *objSAtts* luego de aplicar *augb* a *permSet*.

El primero de ellos indica que el conjunto de objetos a los que un sujeto tiene permiso de acceso luego de una operación de tipo *AugmentingAccess* sólo puede crecer. El segundo certifica que si el sujeto no es el receptor del nuevo permiso o este último no se toma en cuenta entonces todos los objetos del conjunto correspondiente al nuevo estado también estaban en el que corresponde al anterior. Por último, *objSAttsAug_aDef* marca que la versión del conjunto de objetos posterior a la entrega del permiso de acceso está incluido en la versión anterior si esta es ampliada con el objeto sobre el cual se entrega el permiso.

Nuevamente, las pruebas de estos teoremas se encuentran en el apéndice de pruebas pero no serán explicadas y por lo tanto no aparecen en este capítulo.

6.5.3 Teoremas del artículo de BLP

También nos referiremos para los casos válidos al segundo paper [8] de Bell y La-Padula. Allí podemos encontrar dos teoremas que nos servirán para probarlos: el teorema 3.4 y el teorema 3.5:

Teorema 3.4 · Sea v = (b,B,f) seguro y (S,O,$\underline{x}$) $\notin$ b.
Entonces, si b* = b $\cup$ (S,O,$\underline{x}$) y v* = (b*,M,f)

i. Si $\underline{x} = \underline{e}$ ó $\underline{x} = \underline{a}$ ó $\underline{x} = \underline{c}$, v* es seguro.

ii. Si $\underline{x} = \underline{r}$ ó $\underline{x} = \underline{w}$, v* es seguro si y solo si $f_1(S) \geq f_2(O)$ y $f_3(S) \supseteq f_4(O)$

Teorema 3.5 · Sea v = (b,B,f) tal que satisface la propiedad-★ y (S,O,$\underline{x}$) $\notin$ b.
Entonces, si b* = b $\cup$ (S,O,$\underline{x}$) y v* = (b*,M,f)

i. Si $\underline{x} = \underline{e}$ ó $\underline{x} = \underline{c}$, v* satisface la propiedad-★.

ii. Si $\underline{x} = \underline{a}$, v* satisface la propiedad-★ si y solo si
$f_2(O) \geq f_2(O')$ y $f_4(O) \supseteq f_4(O')$ para todo O' $\in$ b(S:r,w).

iii. Si $\underline{x} = \underline{r}$, v* satisface la propiedad-★ si y solo si
$f_2(O) \leq f_2(O')$ y $f_4(O) \subseteq f_4(O')$ para todo O' $\in$ b(S:w,a).

iv. Si $\underline{x} = \underline{w}$, v* satisface la propiedad-★ si y solo si

a. $f_2(O) \geq f_2(O')$ y $f_4(O) \supseteq f_4(O')$ para todo O' $\in$ b(S:r).

b. $f_2(O) \leq f_2(O')$ y $f_4(O) \subseteq f_4(O')$ para todo O' $\in$ b(S:a).

c. $f_2(O) = f_2(O')$ y $f_4(O) = f_4(O')$ para todo O' $\in$ b(S:w).

Por supuesto, podemos esperar que estos dos teoremas valgan en nuestro modelo. Éste sólo difiere de BLP en que son diferentes formalizaciones para los mismos conceptos. A efectos de probar el teorema *GetReadIsSound* sólo serán necesarios el caso *ii* del primero y el caso *iii* del segundo. Sólo definiremos las

partes necesarias para probar los casos A y B de *GetReadIsSound*, que tomando la forma de teoremas separados denominaremos $T34ii$ y $T35iii$, aludiendo a su origen.

6.5.4 Propiedad de Seguridad Simple en un caso válido

Theorem T34ii
$$
\begin{array}{l}
\quad AugmentingAccess \\
\land\ \Xi SecClasses \\
\land\ a \in \{Read, Write\} \\
\land\ ((u, o), a)\ secureRel\ \theta SecClasses \\
\Rightarrow PreservesSimpleSecurity
\end{array}
$$

El teorema $T34ii$ corresponde al caso ii del teorema 3.4 de BLP. Éste enuncia que para cualquier operación que entregue un nuevo permiso de acceso (*AugmentingAccess*) si este es o bien *Read* o bien *Write* y además se da que dicho permiso de acceso es *seguro respecto de la política de seguridad* (*secureRel*), entonces la entrega de este último no introduce un compromiso de seguridad.

Comenzaremos la prueba expandiendo algunas definiciones y luego simplificaremos el resultado, como es usual:

```
invoke AugmentingAccess
invoke PreservesSimpleSecurity
invoke ΔSecClasses
invoke SimpleSecureState
simplify
prenex
```

Luego de las instrucciones anteriores, el objetivo se ha transformado en el siguiente:

$$
\begin{array}{l}
\quad \Delta ActualPermSet \\
\land\ a \in PERMISSION \\
\land\ u \in UID \\
\land\ o \in OID \\
\land\ (u, o) \in \mathrm{dom}\ ActualPermSet \\
\land\ permSet' = augb\ (permSet, (u, o), a) \\
\land\ \Xi SecClasses \\
\land\ a \in \{Read\} \cup \{Write\} \\
\land\ ((u, o), a)\ secureRel\ \theta SecClasses \\
\land\ (\forall\ p\_\_0 : UOp;\ a\_\_1 : PERMISSION \\
\qquad \mid p\_\_0 \in uids \times oids \land a\_\_1 \in permSet\ p\_\_0 \\
\qquad \bullet\ (p\_\_0, a\_\_1)\ secureRel\ \theta SecClasses) \\
\land\ p \in UOp \\
\land\ a\_\_0 \in PERMISSION \\
\land\ p \in uids' \times oids' \\
\land\ a\_\_0 \in permSet'\ p \\
\Rightarrow (p, a\_\_0)\ secureRel\ \theta SecClasses'
\end{array}
$$

Hemos de probar ahora, según el objetivo, que el par $(p, a\_\_0)$ es seguro. El par $(p, a\_\_0)$ es una asociación cualquiera en $permSet'$. Si este último ya existía en $permSet$, entonces por hipótesis vale que es seguro respecto de $SecClasses$, que por no haber cambiado es la misma que $SecClasses'$ y con esto se prueba este caso del teorema.

Sabemos por la definición de $AugmentingAccess$ que sólo el par $((u, o), a)$ se ha agregado a $permSet'$, con lo cual todo par $(p, a\_\_0)$ excepto ese debe estar en $permSet$.

Supongamos entonces dos casos según el par $(p, a\_\_0)$ sea igual a $((u, o), a)$. Separaremos la prueba según valga o no este predicado. Para esto hacemos uso del comando *split*:

split $(p, a\_\_0) = ((u, o), a)$
cases

El comando *split* p convierte el objetivo o en **if** p **then** $p \land o$ **else** $\neg\ p \land o$, y normalmente lo seguiremos de un comando *cases*. Este último particiona el objetivo actual dos o más casos. En el caso del **if** genera dos objetivos, $p \land o$ y $\neg\ p \land o$. Luego de la instrucción anterior se nos presenta el primer caso, en donde vale $(p, a\_\_0) = ((u, o), a)$ y por esto se agrega al objetivo. Así, el teorema queda probado trivialmente por reescritura de $((u, o), a)$ $secureRel\ \theta SecClasses$, resultando en nuestro objetivo $(p, a\_\_0)$ $secureRel\ \theta SecClasses'$. Dejaremos a Z/EVES hacer este trabajo y pasaremos al otro caso:

prove by reduce
next

En el otro caso no vale $(p, a\_\_0) = ((u, o), a)$. Por suspuesto, ya no podemos utilizar el predicado que utilizamos en el caso anterior. El mismo fue incluido especialmente en la operación como una precondición necesaria para que la seguridad se preserve.

Ahora necesitamos utilizar las hipótesis correspondientes a la seguridad en el estado anterior a la operación. Por supuesto, esto implica asumir que $(p, a\_\_0)$ participaba desde antes en la función $permSet$, hecho que no está explicito. Aquí utilizaremos el teoremas auxiliar $augb\_naNoDim$, que asegura que si hay un par en la función $permSet'$ que no es el que fue modificado, entonces éste pertenecía a $permSet$.

Agregaremos el teorema $augb\_naNoDim$ a las premisas con la instrucción *use*:

use $augb\_naNoDim[f := permSet, g := permSet', up := (u, o),$
$na := a, a := a\_\_0]$

rearrange

Como puede verse, para utilizar el teorema *augb_naNoDim* lo instanciamos con los parámetros necesarios. Esto es necesario si los nombres de las variables en el teorema no concuerdan con los de la prueba en curso.

Finalmente, simplificamos algunas condiciones introducidas por *augb_naNoDim* e instanciamos las variables $p\_\_0$ y $a\_\_1$ del cuantificador universal que ya estaba dentro de las hipótesis:

apply *notEqRule* to predicate
    $((u, o), a) \neq (p, a\_\_0)$
apply *tupleInCross2* to predicate
    $((u, o), a) \in (UID \times OID) \times RIGHT$
apply *tupleInCross2* to predicate
    $(p, a\_\_0) \in (UID \times OID) \times RIGHT$
invoke *UOp*
apply *tupleInCross2* to predicate
    $(u, o) \in UID \times OID$
simplify
invoke $\Xi SecClasses$
use *SecClasses$thetasEqual*
instantiate $p\_\_0 == p, a\_\_1 == a\_\_0$
simplify
rearrange
prove by reduce
next
■

Luego de esto Z/EVES se puede encargar de hacer los últimos *modus ponens* y reescrituras necesarias y probar el teorema.

6.5.5 Propiedad-★ en un caso válido

Theorem T35iii
    $AugmentingAccess$
  $\land\ \Xi SecClasses$
  $\land\ a = Read$
  $\land\ (\forall ow : objSAtts\ (\theta ActualPermSet, u, \{ReadWrite, Write\}) \bullet$
      $objclass\ ow\ dominates\ objclass\ o)$
$\Rightarrow\ PreservesStarProperty$

El teorema *T35iii* corresponde al caso *iii* del teorema 3.5 del artículo de BLP, y enuncia que para cualquier operación que entregue el permiso de acceso *Read* a cierto sujeto u con respecto a cierto objeto, si éste satisface la propiedad-★ respecto de los accesos actuales de u, entonces su entrega no introduce un compromiso de seguridad.

Dado que esta prueba es de mayor complejidad que la anterior, la dividiremos en tres teoremas que representan los casos de prueba. Cada teorema está asociado

a un tipo de relación particular entre dos objetos siendo accedidos por cierto usuario en el estado destino. Siendo o el objeto para el que se quiere obtener *Read* y u el usuario que lo requiere, los casos son los siguientes:

Caso 1 · Todos los objetos a los que el sujeto u tiene permiso de escritura tienen una clase que domina la o. Este caso se prueba utilizando una de las guardas definidas en *GetRead*.

Caso 2 · Exceptuando o, todos los objetos a los que el sujeto u tiene permiso de escritura tienen una clase que domina la de todos los objetos a los que u tiene permiso de lectura.

Caso 3 · Todos los sujetos distintos de u validan la propiedad-★.

En las siguientes secciones enunciaremos y probaremos los tres casos y luego los utilizaremos en para probar de $T35iii$.

Caso 1

En este primer caso el sujeto a tratar corresponde al receptor del permiso de acceso. También el objeto afectado se ve involucrado en este teorema. Es el caso más simple ya que la conclusión se deduce casi trivialmente de las hipótesis, en particular de la precondición que es parte de la operación.

Lo enunciaremos de la siguiente manera:

$$
\begin{array}{l}
\textbf{\textit{Theorem}}\ \text{T35iii\_case1} \\
\quad AugmentingAccess \land \Xi SecClasses \land a = Read \land \\
\quad (\forall\, ow : objSAtts\ (\theta ActualPermSet, u, \{ReadWrite, Write\}) \\
\qquad \bullet\ objclass\ ow\ dominates\ objclass\ o) \\
\Rightarrow (\forall\, w : objSAtts\ (\theta ActualPermSet', u, \{ReadWrite, Write\}) \\
\qquad \bullet\ objclass'\ w\ dominates\ objclass'\ o)
\end{array}
$$

Debido a que Z/EVES ya lo presenta en forma prenexa, el cuantificador universal del consecuente desaparece y aparece el predicado que define a w entre las hipótesis. Con eso podemos instanciar la variable ow con w, y eso es lo que haremos.

instantiate $ow == w$

Luego de esto debemos probar que si w está en $objSAtts\ (\theta ActualPermSet' \ldots)$ entonces debe estar también en $objSAtts\ (\theta ActualPermSet \ldots)$, lo cual es cierto por ser el permiso de acceso agregado de tipo *Read*. Esto lo probaremos utilizando uno de los teoremas auxiliares que hemos definido, $objSAttsAug\_na$:

use $objSAttsAug\_na[ou := u, pa := \{Write, ReadWrite\}, x := w]$
rearrange
with predicate $(AugmentingAccess \wedge \exists UsrObj \wedge u \in UID \wedge u \in uids \wedge$
$w \in objSAtts(\theta ActualPermSet', u,$
$(\{Write\} \cup \{ReadWrite\})) \wedge$
$\{Write\} \cup \{ReadWrite\} \in \mathbb{P}\, PERMISSION \wedge$
$(u \neq u \vee a \notin \{Write\} \cup \{ReadWrite\})$
reduce
invoke $\exists SecClasses$
apply $SecClasses\$thetasEqual$ to predicate
$\theta SecClasses = \theta SecClasses'$
reduce
■

Y luego de algunas más reducciones el teorema queda probado.

Caso 2

El segundo caso corresponde probar la propiedad-★ sobre dos objetos que no fueron afectados por la operación, aunque sólo se prueba para el sujeto u. Claramente, usaremos la hipótesis de que valía la propiedad en el estado anterior.

Theorem T35iii_case2
$AugmentingAccess \wedge StarPropertyState \wedge \exists SecClasses \wedge a = Read$
$\Rightarrow (\forall w : objSAtts\ (\theta ActualPermSet', u, \{ReadWrite, Write\});$
$r : objSAtts\ (\theta ActualPermSet', u, \{ReadWrite, Read\}) \mid$
$\neg\ r = o \bullet objclass'\ w\ dominates\ objclass'\ r)$

Comenzaremos la prueba de manera similar a la anterior, pero esta vez instanciando el cuantificador de la definición de *StarPropertyState*. Primero invocaremos la definición de este último y luego instanciaremos las variables con los valores de este caso (u, w, y r).

invoke *StarPropertyState*
instantiate $u\_\_0 == u, ow == w, or == r$

Aquí se nos presenta un objetivo con la siguiente forma:

$$
\begin{array}{l}
\vdots \\
\wedge\ w \in objSAtts\ (\theta ActualPermSet, u, \{ReadWrite, Write\}) \\
\quad \wedge\ r \in objSAtts\ (\theta ActualPermSet, u, \{ReadWrite, Read\}) \\
\quad\ \vdots \\
\quad\ \Rightarrow objclass\ w\ dominates\ objclass\ r \\
\\
\wedge\ w \in objSAtts\ (\theta ActualPermSet', u, \{ReadWrite, Write\}) \\
\wedge\ r \in objSAtts\ (\theta ActualPermSet', u, \{ReadWrite, Read\}) \\
\wedge \neg\ r = o \\
\vdots \\
\Rightarrow objclass'\ w\ dominates\ objclass'\ r
\end{array}
$$

Comenzaremos instanciando el teorema *objSAttsAug_na*. Lo usaremos para probar $w \in objSAtts\ (\theta ActualPermSet, u, \{ReadWrite, Write\})$, ya que alude a un caso en el que no interfiere la operación. Esto no se da en el término análogo para r, y por lo tanto utilizaremos otro del mismo grupo de teoremas: *objSAttsAug_aDef*.

use *objSAttsAug_na*
[$pa := \{ReadWrite, Write\}, ou := u, x := w$]
use *objSAttsAug_aDef*
[$pa := \{ReadWrite, Read\}, x := r$]
⋮
■

Para utilizarlos, como es usual, hacemos ciertas manipulaciones del objetivo con instrucciones *rearrange* y *simplify* entre otras. Luego de esto los consecuentes de los teoremas quedan incluidos directamente dentro de nuestras hipótesis y en pocos pasos más culminamos la prueba.

Caso 3

En el último de los casos necesarios para probar $T35iii$ el sujeto tratado no fue afectado por la operación *GetRead*:

$$
\begin{array}{l}
\textbf{\textit{Theorem}}\ \text{T35iii\_case3} \\
\quad AugmentingAccess \wedge StarPropertyState \wedge \Xi SecClasses \wedge a = Read \\
\quad \Rightarrow (\forall\ ou : uids \mid \neg\ ou = u \\
\qquad \bullet\ (\forall\ w : objSAtts\ (\theta ActualPermSet', ou, \\
\qquad\qquad \{ReadWrite, Write\}); \\
\qquad\quad r : objSAtts\ (\theta ActualPermSet', ou, \\
\qquad\qquad \{ReadWrite, Read\}) \\
\qquad\quad \bullet\ objclass'\ w\ dominates\ objclass'\ r\ ))
\end{array}
$$

Debido a que el sujeto no fue afectado por la operación, tampoco lo fue ninguno de los dos conjuntos de objetos involucrados en el caso. Por lo tanto, usaremos *objSAttsAug_na* tanto sobre w como sobre r.

use $objSAttsAug\_na[pa := \{ReadWrite, Write\}, x := w]$
use $objSAttsAug\_na[pa := \{ReadWrite, Read\}, x := r]$
⋮

Y luego de algunas aplicaciones de reglas sintácticas, simplificaciones, y reducciones, llegamos a que valen los consecuentes de los teoremas recién utilizados: $w \in objSAtts\ (\theta ActualPermSet, \ldots)$ y $r \in objSAtts\ (\theta ActualPermSet, \ldots)$. Esto nos posibilita instanciar el cuantificador de *StarPropertyState* con ou, w, y r para entonces concluir la prueba con un *simplify*.

instantiate $u\_\_0 == ou, ow == w, or == r$
⋮
simplify
■

Prueba

Finalmente, habiendo probado los tres casos, probaremos el teorema $T35iii$. Con la ayuda de los teoremas anteriores la prueba de este último resulta mucho más simple y se reduce a dividir la prueba en los casos ya probados para poder aplicarlos. Comenzaremos la prueba con las invocaciones usuales y luego dividiremos la prueba en dos casos, según valga $(u\_\_0, or) = (u, o)$:

invoke *PreservesStarProperty*
invoke $\Delta SecClasses$
invoke *StarPropertyState*
simplify
prenex
split $u\_\_0 = u \wedge or = o$
cases

Ahora podemos aplicar el primero de los teoremas, $T35iii\_case1$:

use $T35iii\_case1[w := ow]$
rearrange
reduce
next

El caso queda probado con una instrucción *reduce* y pasamos al siguiente. Ahora dividiremos nuevamente según valga $u\_\_0 = u$.

split $u\_0 = u$
cases
simplify

Luego de esto tenemos que $u\_0 = u$ y $or = o$, escenario en el cual se aplica el segundo caso:

use $T35iii\_case2[r := or, w := ow]$
rearrange
simplify
invoke *StarPropertyState*
simplify
reduce
next

Una vez más, son pocas las instrucciones necesarias para probarlo y pasamos al tercero.

simplify
use $T35iii\_case3[r := or, w := ow, ou := u\_0]$
rearrange
simplify
reduce
next
∎

Al igual que en los anteriores, utilizamos el teorema correspondiente y con esto terminamos la prueba del teorema $T35iii$.

6.6 Certificación de GetRead

Ya tenemos todo lo necesario para probar el teorema final de este trabajo. Delinearemos la estrategia de prueba, y por último daremos el resultado final.

Una vez más, de la misma forma que para la prueba del teorema $T35iii$, especificaremos y probaremos los casos propuestos en la sección 6.3.1 utilizando los teoremas que probamos a lo largo todas las secciones anteriores: *SimpleSecurityNotAffected*, *StarPropertyNotAffected*, $T34ii$ y $T35iii$.

Probaremos los cuatro casos de *GetReadIsSound* en grupos de dos, según un factor común entre ellos. Los casos en los que la operación se rechaza por un lado y los casos en los que la operación tiene éxito por otro.

6.6.1 En un caso de rechazo

El primero y más trivial de estos dos teoremas es *GetReadRejectIsSound*, cuyo enunciado y prueba siguen:

Theorem GetReadRejectIsSound
$\forall$ *GetReadReject* •
GetReadNotAffected $\Rightarrow$ *PreservesSimpleSecurity* $\wedge$
PreservesStarProperty

proof)
apply *SimpleSecurityNotAffected*
apply *StarPropertyNotAffected*
prove
■

El consecuente es implicado casi trivialmente por *SimpleSecurityNotAffected* y *StarPropertyNotAffected* teniendo *GetReadReject* y *GetReadNotAffected* por hipótesis, y el teorema queda probado.

6.6.2 En un caso válido

El segundo, menos trivial, es el teorema *GetReadOkYesIsSound*, cuyo enunciado es el siguiente:

Theorem GetReadOkYesIsSound
$\forall$ *GetReadOkYes* •
GetReadNotAffected $\Rightarrow$ *PreservesSimpleSecurity* $\wedge$
PreservesStarProperty

Probaremos la propiedad de seguridad simple y la propiedad-★ por separado. Utilizando *cases*, se nos presenta primero el caso de *PreservesSimpleSecurity*, donde utilizamos el teorema $T34ii$:

cases
invoke *GetReadOkYes*
invoke *GetReadNotAffected*
use $T34ii[o?/o, u?/u, a := Read]$

El enunciado del teorema $T34ii$ no es sintácticamente compatible con el enunciado del objetivo en este paso. Haremos algunas manipulaciones sintácticas:

apply *secureRel_Def* to predicate
$((u?, o?), Read) secureRel \theta SecClasses$
with predicate $(\theta SecClasses \in SecClasses$
$\land\ u? \in \theta SecClasses.uids$
$\land\ o? \in \theta SecClasses.oids$
$\land\ Read \in PERMISSION)$
reduce
with predicate $(Read \in \{Read\} \cup \{ReadWrite\})$
reduce
with predicate (
$\theta SecClasses.usrclass\ u?\ dominates\ \theta SecClasses.objclass\ o?)$
reduce
with predicate $(Read \in \{Read\} \cup \{Write\})$
reduce
simplify
next

Luego de *simplify* el primer caso queda probado y pasamos al segundo. En este último caso utilizamos el teorema $T35iii$:

use $T35iii[o?/o, u?/u, a := Read]$
rearrange
simplify
invoke *GetReadOkYes*
with predicate (
$\forall\ ow : objSAtts(\theta ActualPermSet, u?, (\{ReadWrite\} \cup \{Write\}))$
$\bullet\ objclass\ ow\ dominates\ objclass\ o?)$
rewrite
simplify
next
∎

Siguen algunas simplificaciones necesarias y el teorema queda probado luego de la última simplificación.

6.6.3 Consistencia total

Habiendo probado los dos casos, la prueba de *GetReadIsSound* resulta directa:

Theorem GetReadIsSound
$\forall\ GetRead \bullet PreservesSimpleSecurity \land PreservesStarProperty$

proof)
invoke *GetRead*
split *GetReadOkYes*
cases
use *GetReadOkYesIsSound*
simplify
next

use *GetReadRejectIsSound*
simplify
next
∎

La prueba de *GetReadIsSound*, la más trascendente de las pruebas de este trabajo, demuestra que la operación *GetRead* preserva las propiedades que constituyen la política de seguridad. De esta forma, confirmamos formalmente que su ejecución no introduce un compromiso de seguridad.

6.7 Pruebas de chequeo de dominio

Los chequeos de dominio son un mecanismo de control de Z/EVES 2.1 que ayudar al usuario a escribir especificaciones correctas, yendo un poco más allá del análisis sintáctico inicial [13]. Se procede con los chequeos para todo término de la especificación en busca de aplicaciones de funciones parciales a fin de verificar que la aplicación siempre ocurre sobre el dominio de la función.

La mayoría de estos chequeos son verificados de manera automática. Sin embargo, en los casos más difíciles se generan teoremas cuyas pruebas son responsabilidad del usuario. Este último debe usar el mecanismo de prueba estándar para certificar que efectivamente valen. Para esta especificación se han probado cerca de una decena de pruebas de chequeo de dominio, todas las involucradas en las pruebas relacionadas con la consistencia de *GetRead*. Estas pruebas fueron incluidas en el apéndice C. Los teoremas tienen nombres como *Entidad*$*DomainCheck*, donde *Entidad* es el nombre de la entidad cuya definición se verifica.

Capítulo 7

Aplicación

Entre otras cosas, la especificación que hemos desarrollado permite analizar mejor las aplicaciones clientes del módulo. Basándonos en la especificación del módulo podemos verificar, a su vez, las propiedades de estas últimas. Si deseamos especificar formalmente las aplicaciones clientes o sus operaciones, entonces podemos utilizar las especificación de aquél para verificar las propiedades deseadas.

Daremos el ejemplo de una operación en una aplicación cliente del módulo sobre la que queremos verificar la preservación de seguridad. Para esto especificaremos una operación que interactúa con el módulo, utilizando la definición de las operaciones antes dadas como base. Así podremos razonar sobre las partes componentes de la operación para probar lo deseado. En este proceso intervienen también los teoremas desarrollados en la verificación de *GetRead*.

7.1 Un caso de uso

En el contexto de un servidor de aplicaciones web, supóngase el ejemplo de un requerimiento que permita listar las entradas de cierta tabla en una base de datos en un formato dado por un archivo de configuración de reportes y que luego de ejecutar la operación se añada una entrada a un archivo de bitácora que registra todas las operaciones del sistema.

Este podría ser un pedido HTTP que tiene por objeto ejecutar el proceso recién descripto y llevar el resultado, es decir, el listado, al cliente a través de una respuesta HTTP (o bien información acerca de porqué no se puede ejecutar la operación requerida).

Utilizaremos este ejemplo porque representa una operación en la que se interactúa con varias entidades de diferentes modos. Así se ve como se aplica la política de seguridad definida por el módulo a una variedad de entidades que se administran en forma segura dentro de la aplicación.

Analizemos la funcionalidad de la operación *listado*. Se desean cuatro accesos relevantes a la seguridad del sistema en esta operación:

- acceso en modo lectura al archivo de configuración.
- acceso en modo escritura al archivo de bitácora.
- acceso en modo lectura a la tabla de la base de datos asociada a la operación.
- acceso en modo escritura la entidad que representa la respuesta HTTP a ser enviada a la terminal del cliente.

Tendremos que obtener del módulo los permisos de acceso para efectuar estas operaciones, y al finalizar los liberaremos.

Es importante indentificar todas las entidades relevantes a la seguridad con las que se interactúa en cada caso. Para cada una de estas entidades se debe tener su *OID* asociado de tal forma de poder pedir el acceso al módulo de control de acceso. Para esto, como hemos mencionado anteriormente, es necesario haber registrado cada entidad en el módulo de control de acceso en una etapa anterior de procesamiento.

Asumimos que la ejecución de la operación sólo requiere de esos modos de acceso. Por simplicidad, es nuestra asunción que el resto de las sub-operaciones ejecutadas entre las que piden los permisos de acceso y las que los liberan se asumen que no alteran el estado del módulo de control de acceso.

7.2 Análisis de preservación de seguridad

Supongase que queremos añadir esta operación al sistema y queremos verificar que el sistema se mantiene seguro. Entonces será necesario un análisis formal con una prueba de que la operación no introduce un compromiso de seguridad.

Naturalmente haremos esto de la misma forma que se hizo con las operaciones del módulo. Especificaremos el comportamiento de la operación que es relevante a la seguridad del sistema en Z y luego definiremos los predicados que serán necesarios probar para certificar que el sistema se mantiene seguro.

7.2.1 Especificación

$$
\begin{aligned}
listado \mathrel{\widehat=} & \; GetRead[o? := Configuracion.OID] \fatsemi \\
& GetRead[o? := Tabla.OID] \fatsemi \\
& GetWrite[o? := Log.OID] \fatsemi \\
& GetWrite[o? := Respuesta.OID] \fatsemi \\
& ObtenerConfiguracion \fatsemi \\
& ImprimirListado \fatsemi \\
& EscribirBitacora \fatsemi \\
& Release[o? := Respuesta.OID, a? := Write] \fatsemi \\
& Release[o? := Log.OID, a? := Write] \fatsemi \\
& Release[o? := Tabla.OID, a? := Read] \fatsemi \\
& Release[o? := Configuracion.OID, a? := Read]
\end{aligned}
$$

Para la especificación de esta operación hemos utilizado un operador disponible en Z que no habíamos utilizado hasta ahora, la composición de esquemas: $\fatsemi$.

El operador $\fatsemi$ toma dos esquemas y produce un nuevo esquema en el que las variables primadas del primero se utilizan en el segundo como no primadas, y las primadas del segundo representan el resultado de la operación. De esta forma Z modela secuencia; $A \fatsemi B$ significa procesar el esquema A primero y luego B sobre la salida de A.

Las variables de entrada que reciben el identificador de objeto son instanciadas con los correspondientes. Las variables que reciben el identificador de sujeto, debido a que todas corresponden al mismo, son compartidas. El sujeto es, por ende, especificado a través de la variable $u?$.

Como hemos asumido que las operaciones intermedias (*ObtenerConfiguracion*, *ImprimirListado*, y *EscribirBitacora*) no modifican el estado del módulo de control de acceso, a efectos de un análisis de seguridad es equivalente especificar esta operación en forma simplificada:

$$
\begin{aligned}
listado \mathrel{\widehat=} \; & GetRead[o? := Configuracion.OID]_{\S} \\
& GetRead[o? := Tabla.OID]_{\S} \\
& GetWrite[o? := Log.OID]_{\S} \\
& GetWrite[o? := Respuesta.OID]_{\S} \\
& Release[o? := Respuesta.OID, a? := Write]_{\S} \\
& Release[o? := Log.OID, a? := Write]_{\S} \\
& Release[o? := Tabla.OID, a? := Read]_{\S} \\
& Release[o? := Configuracion.OID, a? := Read]
\end{aligned}
$$

De la misma manera, los objetos *Configuracion*, *Tabla*, *Log*, y *Respuesta* son utilizados sólo para obtener sus identificadores (*OID*), por lo que equivale, a efectos de nuestro análisis, definir simplemente sus *OID*s como parámetros de entrada:

$$
\begin{aligned}
listado \mathrel{\widehat=} \; & GetRead[configuracionOID?/o?]_{\S} \\
& GetRead[tablaOID?/o?]_{\S} \\
& GetWrite[logOID?/o?]_{\S} \\
& GetWrite[respuestaOID?/o?]_{\S} \\
& Release[respuestaOID?/o?, a? := Write]_{\S} \\
& Release[logOID?/o?, a? := Write]_{\S} \\
& Release[tablaOID?/o?, a? := Read]_{\S} \\
& Release[configuracionOID?/o?, a? := Read]
\end{aligned}
$$

Notar que no hemos incluido manejo de errores en esta operación, si fuera así la estructura sería más complicada ya que para aquellas operaciones que aceptar fallo como *GetRead* o *GetWrite* es necesario determinar si la operación falló para no continuar con la transacción. Si el lenguaje de implementación tiene excepciones entonces también sería posible evitar la complejidad delegando parte del comportamiento de manejo de error en éstas. De cualquier forma, no afecta en profundidad al análisis que queremos mostrar aquí y por lo tanto el manejo de casos de error no se mostrará.

7.2.2 Verificación

Para verificar que la operación recién definida preserva seguridad debemos hacer lo mismo que se hizo para *GetRead* en forma análoga:

Theorem ListadoIsSound
$\forall Listado \bullet PreservesSimpleSecurity \land PreservesStarProperty$

El hecho más sustancial a analizar en la prueba de esta operación es que la misma está en su totalidad definida en terminos de la composición de operaciones seguras.

Notar que, de hecho, la composición de operaciones que preservan seguridad preserva seguridad. Es decir, si tanto A como B preservan seguridad y propiedad-$\bigstar$ entonces $A \fatsemi B$ también las preserva.

Debido a aspectos técnicos de Z/EVES 2.1 un teorema así no resulta sencillo, por lo que nos vemos obligados a dividir la definición de *Listado* en varias partes agrupando siempre de a pares las instrucciones para poder aplicar fácilmente el teorema de la preservación de seguridad a través de la secuencia de operaciones.

Como este procedimiento excede en esfuerzo nuestro objetivo de ejemplificar el procedimiento de aplicación, no desarrollaremos en mayor detalle esta prueba.

Capítulo 8

Conclusión

Este trabajo tiene dos resultados fundamentales. Un primer resultado es que la especificación/verificación, aunque parcial, ha permitido un análisis de consistencia de profundidad evidente de los conceptos utilizados. Implementaciones del módulo pueden ofrecer un alto grado de confiabilidad si se basan en este trabajo, y sobre todo si éste es concluido para el resto de las operaciones. Técnicas como *refinamiento* pueden utilizarse para desarrollar implementaciones correctas a partir de esta especificación [13, 15].

Un segundo resultado es que ésta constituye una base fiable sobre la cual se pueden diseñar y construir aplicaciones clientes del módulo. El procedimiento descrito en el capítulo 7 se aplica a cualquier operación que interactúe con el módulo de control de acceso. Operaciones especificadas en términos de las operaciones que constituyen la interfaz del módulo pueden usar los teoremas desarrollados en su análisis de preservación de seguridad correspondiente. Transmitiendo así la confiabilidad que se desarrollo para el módulo a las aplicaciones clientes.

La especificación de la operación *GetRead* y la verificación de su preservación de la seguridad constituye un ejemplo claro de técnicas formales aplicadas al desarrollo de software con propiedades de seguridad de importancia crítica.

La confiabilidad de los estrictos mecanismos lógicos en los que se basan las pruebas se transmite a este trabajo a través de estas últimas.

Continuación de este trabajo

Debido a que excede en esfuerzo los objetivos de este trabajo, no se ha verificado la consistencia para todas las operaciones especificadas. Además, algunas de las operaciones sugeridas fueron omitidas en la especificación. De continuarse este trabajo, el primer paso sería especificar las operaciones restantes y luego de esto probar la consistencia para todas ellas, además de las pruebas de chequeo de dominio para el resto de las entidades.

Finalmente se puede certificar la seguridad a nivel módulo. Al respecto Bell y LaPadula enunciaron el Teorema Básico de Seguridad (BST, Basic Security Theorem), que define las propiedades que un sistema debe tener para ser seguro por BLP. Aunque algo cuestionado [11], constituye la base teórica de la seguridad a nivel sistema en el modelo BLP [8]. Según este teorema, la politica de seguridad queda certificada si todas las operaciones preservan seguridad y el estado inicial la certifica. Un equivalente a este teorema puede ser desarrollado para esta especificación. Aplicando la lógica de Z/EVES a esta última verificación se puede probar la consistencia a nivel módulo con la misma confiabilidad que se ha probado la consistencia de la operación *GetRead*.

Apéndice A

Designaciones

Las designaciones son una técnica que ayuda a la autoexplicación de una especificación asociando cada entidad con una descripción en lenguaje natural del concepto que abstrae. Por notación se suele usar el símbolo $\approx$, exhibiendo a su lado izquierdo el nombre de la entidad designada y del lado derecho la designación.

Hemos incluido las designaciones de las entidades esenciales de nuestra especificación en la siguiente lista. Para los conceptos compuestos o con algún tipo de estructura se designan además sus partes de manera indentada, como es el caso de la entidad *MESSAGE*.

UID $\approx$ Identificadores de sujeto posibles.

OID $\approx$ Identificadores de objeto posibles.

CATEGORY $\approx$ Categorías posibles para las etiquetas de seguridad.

LEVEL $\approx$ Niveles posibles para las etiquetas de seguridad.

MESSAGE $\approx$ Mensajes del sistema (en este caso todos errores, podría haber otros).

- *ObjDoesNotExist* $\approx$ Mensaje que indica que el objeto en referenciado no existe.
- *InvalidUser* $\approx$ Mensaje que indica que el sujeto indicado es inválido.
- *ObjAlreadyExists* $\approx$ Mensaje que indica que el sujeto referenciado ya existe en el sistema.

FUNCTION $\approx$ El conjunto de todos los identificadores de funciones.

fGetRead $\approx$ Identifica a la función *GetRead* del módulo.
fGetWrite $\approx$ Identifica a la función *GetWrite* del módulo.
fGetReadWrite $\approx$ Identifica a la función *GetReadWrite*.
fGetExecute $\approx$ Identifica a la función *GetExecute*.
fRelease $\approx$ Identifica a la función *Release* del módulo.
fGive $\approx$ Identifica a la función *Give* del módulo.
fRescind $\approx$ Identifica a la función *Rescind* del módulo.
fChangeObjClass $\approx$ Identifica a la función *ChangeObjClass*.
fCreateObject $\approx$ Identifica a la función *CreateObject*.
fDeleteObject $\approx$ Identifica a la función *DeleteObject*.

EXCEPTION $\approx$ El conjunto de excepciones de ejecución posibles del sistema.

DECISION $\approx$ Decisiones que toma el sistema ante un pedido de un sujeto.

YES $\approx$ La operación se ejecutó satisfactoriamente.
NO $\approx$ El permiso para ejecutar la operación fue denegado.
mkerr $\approx$ Indica que la operación tuvo un error de ejecución.

ACCESS_ATTRIBUTE $\approx$ Conjunto de atributos de acceso ("modos de acceso") del sistema.

Read $\approx$ Permiso de acceso de lectura.
Write $\approx$ Permiso de acceso de escritura.
ReadWrite $\approx$ Permiso de acceso de lectura y escritura.
Execute $\approx$ Permiso de acceso de ejecución.
Control $\approx$ Permiso de acceso de control.

RIGHT $\approx$ Derechos de acceso del sistema.

PERMISSION $\approx$ Permisos de acceso del sistema.

fUC $\approx$ Tipo de las funciones de sujeto en clases de seguridad.

fOC $\approx$ Tipo de las funciones de objeto en clases de seguridad.

SecClass $\approx$ Clase de seguridad.

level $\approx$ El nivel de una clase de seguridad.
categories $\approx$ El conjunto de categorías de una clase de seguridad.

BottomSecClass $\approx$ Clase de seguridad de valor mínimo, es dominada por cualquier otra.

TopSecClass $\approx$ Clase de seguridad de valor máximo, domina a cualquier otra.

InitOSecClass $\approx$ Clase de seguridad que se le asigna a un objeto recién creado.

dominates $\approx$ Relación entre dos clases de seguridad que indica que la de la izquierda domina a la de la derecha.

matches $\approx$ Relación entre dos clases de seguridad que indica que son equivalentes.

Administrator $\approx$ Usuario especial con privilegios superiores.

UsrObj $\approx$ Parte del estado del sistema que describe los sujetos y objetos existentes.

- *uids* $\approx$ El conjunto de identificadores de sujeto en uso en el sistema.
- *oids* $\approx$ El conjunto de identificadores de objeto en uso en el sistema.

SecClasses $\approx$ Parte del estado del sistema en la que se definen las clases de seguridad de los objetos y sujetos.

- *usrclass* $\approx$ Dado un sujeto, retorna su clase de seguridad.
- *objclass* $\approx$ Dado un objeto, retorna su clase de seguridad.

augb $\approx$ Agrega a una función de tipo *fUO_PERMS* un nuevo permiso a un par (sujeto,objeto).

dimb $\approx$ Quita a una función de tipo *fUO_PERMS* un permiso a un par (sujeto,objeto).

secureRel $\approx$ Relación que indica qué ternas sujeto, objeto, atributo satisfacen la condición de seguridad relativa a una clasificación dada.

ACLS $\approx$ Parte del estado del sistema que mantiene las listas de control de acceso.

- *acl* $\approx$ Dado un objeto, da la lista de control de acceso de dicho objeto.

MACAdmin $\approx$ Parte del estado del modulo que mantiene información acerca de los administradores de la politica MAC.

- *macAdmins* $\approx$ Conjunto de usuarios con derecho de administrar la politica MAC.

ActualPermSet $\approx$ Parte del estado del sistema que mantiene información acerca de los permisos de acceso actuales.

permSet $\approx$ Dado un objeto y un sujeto, da los permisos de acceso que posee dicho sujeto sobre el objeto.

objSAtts $\approx$ Función que dado un sujeto y un conjunto de permisos de acceso da la colección de objetos para los que el sujeto tiene alguno de dichos permisos de acceso.

SimpleSecureState $\approx$ Esquema en el cual las condiciones para un estado seguro de sistema se satisfacen.

PreservesSimpleSecurity $\approx$ Esquema en el cual las condiciones para una que regla preserve seguridad se satisfacen.

StarPropertyState $\approx$ Estado en el cual se satisface la propiedad-★.

PreservesStarProperty $\approx$ Operación que preserva la propiedad-★.

AugmentingAccess $\approx$ Operación que entrega cierto permiso de acceso.

ErrorObjDoesNotExist $\approx$ Error de ejecución en el cual cierto objeto no existe.

ErrorInvalidUser $\approx$ Error de ejecución causado por una referencia a un usuario inválido.

Error $\approx$ Error de ejecución en el módulo de control de acceso.

function $\approx$ Función asociada en la que se encontró la excepcion.

message $\approx$ El tipo de error encontrado en la excepción.

SystemState $\approx$ Representa el estado del módulo de control de acceso.

Apéndice B

Especificación Completa

Este capítulo presenta la especificación completa desarrollada para este trabajo. Incluye tanto lo ya mostrado en el capítulo 5 como lo omitido.

La siguiente lista muestra todas las entidades en orden de dependencia, es decir, primero las más basicas, y luego aquellas que se construyen a partir de las anteriores.

$[UID]$

$[OID]$

$[CATEGORY]$

$LEVEL == \mathbb{N}$

$MESSAGE ::= ObjDoesNotExist \mid ObjAlreadyExists \mid InvalidUser$

$$
\begin{aligned}
FUNCTION ::= \; & fGetRead \\
\mid \; & fGetWrite \\
\mid \; & fGetReadWrite \\
\mid \; & fGetExecute \\
\mid \; & fRelease \\
\mid \; & fGive \\
\mid \; & fRescind \\
\mid \; & fChangeObjClass \\
\mid \; & fCreateObject \\
\mid \; & fDeleteObject
\end{aligned}
$$

$$EXCEPTION == MESSAGE \times FUNCTION$$

$$DECISION ::= YES \mid NO \mid mkerr \langle\langle EXCEPTION \rangle\rangle$$

$$ACCESS\_ATTRIBUTE ::= Read \mid Write \mid ReadWrite \mid Execute \mid Control$$

$$RIGHT == ACCESS\_ATTRIBUTE$$

$$PERMISSION == ACCESS\_ATTRIBUTE \setminus \{Control\}$$

SecClass

$$level : LEVEL$$
$$categories : \mathbb{P}\, CATEGORY$$

BottomSecClass

$$SecClass$$

$$categories = \emptyset$$
$$level = \min(LEVEL)$$

TopSecClass

$$SecClass$$

$$categories = CATEGORY$$
$$level = \max(LEVEL)$$

$$InitOSecClass \mathrel{\widehat=} BottomSecClass$$

syntax *dominates inrel*

$$\_dominates\_ : SecClass \leftrightarrow SecClass$$

$$\forall x, y : SecClass \bullet$$
$$x \ dominates \ y \Leftrightarrow$$
$$(y.level \leq x.level \land y.categories \subseteq x.categories)$$

syntax *matches inrel*

$$\_matches\_ : SecClass \leftrightarrow SecClass$$

$$\forall x, y : SecClass \bullet$$
$$x \ matches \ y \Leftrightarrow$$
$$(y.level = x.level \land y.categories = x.categories)$$

$$fUC == UID \nrightarrow SecClass$$

$$fOC == OID \nrightarrow SecClass$$

$$Administrator : UID$$

UsrObj

$$oids : \mathbb{P}\, OID$$
$$uids : \mathbb{P}\, UID$$

$$Administrator \in uids$$

SecClasses

$$UsrObj$$
$$usrclass : fUC$$
$$objclass : fOC$$

$$\text{dom}\ usrclass = uids$$
$$\text{dom}\ objclass = oids$$

MACAdmin

$$UsrObj$$
$$macAdmins : UID$$

$$macAdmins \subseteq uids$$
$$macAdmins \neq \emptyset$$

$$UOp == UID \times OID$$

$$fUO\_RIGHT == UOp \nrightarrow \mathbb{P}\, RIGHT$$

$$fUO\_PERMS == UOp \nrightarrow \mathbb{P}\, PERMISSION$$

$$augb : fUO\_PERMS \times UOp \times PERMISSION \nrightarrow fUO\_PERMS$$

$$\text{dom}\ augb = \{f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$$
$$up \in \text{dom}\, f\}$$
$$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$$
$$up \in \text{dom}\, f \bullet augb\ (f, up, na) = f \oplus \{(up \mapsto fup \cup \{na\})\}$$

$$dimb : fUO\_PERMS \times UOp \times PERMISSION \rightarrow fUO\_PERMS$$

$$\begin{aligned}&\operatorname{dom} dimb = \{f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid \\ &\qquad up \in \operatorname{dom} f\} \\ &\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid up \in \operatorname{dom} f \\ &\quad \bullet\ dimb\ (f, up, na) = f \oplus \{(up \mapsto fup \setminus \{na\})\}\end{aligned}$$

syntax *secureRel inrel*

$$\_secureRel\_ : UOp \times PERMISSION \leftrightarrow SecClasses$$

$$\begin{aligned}&\forall f : SecClasses \bullet \\ &\quad \forall u : f.uids;\ o : f.oids;\ a : PERMISSION \bullet \\ &\qquad ((u, o), a)\ secureRel\ f \Leftrightarrow \\ &\qquad\quad a \in \{Read, ReadWrite\} \Rightarrow \\ &\qquad\qquad f.usrclass\ u\ dominates\ f.objclass\ o)\end{aligned}$$

ACLS

$$\begin{aligned}&UsrObj \\ &acl : OID \nrightarrow \mathbb{P}(UID \times RIGHT) \\ &\operatorname{dom} acl = oids\end{aligned}$$

ActualPermSet

$$\begin{aligned}&UsrObj \\ &permSet : fUO\_PERMS \\ &\operatorname{dom} permSet = uids \times oids\end{aligned}$$

$$objSAtts : ActualPermSet \times UID \times \mathbb{P}\ PERMISSION \nrightarrow \mathbb{P}\ OID$$

$$\begin{aligned}&\operatorname{dom} objSAtts = \{m : ActualPermSet;\ u : UID; \\ &\qquad atts : \mathbb{P}\ PERMISSION \mid u \in m.uids\} \\ &\forall m : ActualPermSet \bullet \\ &\quad \forall u : m.uids;\ atts : \mathbb{P}\ PERMISSION \bullet \\ &\qquad objSAtts\ (m, u, atts) \subseteq m.oids \\ &\forall m : ActualPermSet \bullet \\ &\quad \forall u : m.uids;\ atts : \mathbb{P}\ PERMISSION \bullet \\ &\qquad objSAtts\ (m, u, atts) = \\ &\qquad\qquad \{o : m.oids;\ a : atts \mid a \in m.permSet\ (u, o) \bullet o\}\end{aligned}$$

SimpleSecureState

$$\begin{aligned}&ActualPermSet \\ &SecClasses \\ &\forall p : UOp;\ a : PERMISSION \mid \\ &\quad p \in uids \times oids \wedge a \in permSet\ p \\ &\quad \bullet\ (p, a)\ secureRel\ \theta SecClasses\end{aligned}$$

$$PreservesSimpleSecurity \mathrel{\widehat=} \Delta ActualPermSet \land \Delta SecClasses \land (SimpleSecureState \Rightarrow SimpleSecureState')$$

StarPropertyState

$ActualPermSet$
$SecClasses$

$\forall u : UID;\ ow, or : OID \mid u \in uids \land$
$ow \in objSAtts\ (\theta ActualPermSet, u, \{Write, ReadWrite\}) \land$
$or \in objSAtts\ (\theta ActualPermSet, u, \{Read, ReadWrite\})$
$\bullet$ *objclass ow dominates objclass or*

$$PreservesStarProperty \mathrel{\widehat=} \Delta ActualPermSet \land \Delta SecClasses \land (StarPropertyState \Rightarrow StarPropertyState')$$

AugmentingAccess

$\Delta ActualPermSet$
$a : PERMISSION$
$u : UID$
$o : OID$

$(u, o) \in \mathrm{dom}\ permSet$
$permSet' = augb\ (permSet, (u, o), a)$

Error

$function : FUNCTION$
$message : MESSAGE$
$out! : DECISION$

$out! = mkerr\ (message, function)$

ErrorObjDoesNotExist

$UsrObj$
$o? : OID$
$u? : UID$
$Error[message := ObjDoesNotExist]$

$o? \notin oids \land u? \in uids$

ErrorInvalidUser

$UsrObj$
$u? : UID$
$Error[message := InvalidUser]$

$u? \notin uids$

GetReadOkYes

$$
\begin{array}{l}
AugmentingAccess[a := Read, u?/u, o?/o] \\
SecClasses \\
ACLS \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \\
(u?, Read) \in acl\ o? \\
usrclass\ u?\ dominates\ objclass\ o? \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Write, ReadWrite\}) \\
\quad \bullet\ objclass\ o\ dominates\ objclass\ o? \\
out! = YES
\end{array}
$$

GetReadOkNo

$$
\begin{array}{l}
SecClasses \\
ACLS \\
ActualPermSet \\
u? : UID \\
o? : OID \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \\
out! = NO \\
\neg\ ((u?, Read) \in acl\ o? \\
\quad \land\ usrclass\ u?\ dominates\ objclass\ o? \\
\quad \land\ (\exists o : objSAtts\ (\theta ActualPermSet, u?, \{Write, ReadWrite\}) \\
\qquad \bullet \neg\ objclass\ o\ dominates\ objclass\ o?)\ )
\end{array}
$$

$$GetReadEObjNotExist \mathrel{\widehat=} ErrorObjDoesNotExist[function := fGetRead]$$

$$GetReadEInvalidUser \mathrel{\widehat=} ErrorInvalidUser[function := fGetRead]$$

$$SystemState \mathrel{\widehat=} UsrObj \land SecClasses \land ACLS \land ActualPermSet$$

$$
\begin{array}{l}
GetReadNotAffected \mathrel{\widehat=} \Xi UsrObj \land \\
\qquad \Xi SecClasses \land \\
\qquad \Xi ACLS \land \\
\qquad \Xi MACAdmin
\end{array}
$$

$$
\begin{array}{l}
GetReadReject \mathrel{\widehat=} \Xi ActualPermSet \land (GetReadOkNo \lor \\
\qquad GetReadEInvalidUser \lor \\
\qquad GetReadEObjNotExist)
\end{array}
$$

$$
\begin{array}{l}
GetRead \mathrel{\widehat=} GetReadNotAffected \land (GetReadOkYes \lor \\
\qquad GetReadReject)
\end{array}
$$

GetWriteOkYes

$AugmentingAccess[u?/u, o?/o, a := Write]$
$SecClasses$
$ACLS$
$out! : DECISION$

$o? \in oids \land u? \in uids$
$(u?, Write) \in acl\ o?$
$\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Read, ReadWrite\})$
$\bullet\ objclass\ o?\ dominates\ objclass\ o$
$out! = YES$

GetWriteOkNo

$ActualPermSet$
$SecClasses$
$ACLS$
$u? : UID$
$o? : OID$
$out! : DECISION$

$o? \in oids \land u? \in uids$
$out! = NO$
$(u?, Write) \notin acl\ o?$
$\lor (\exists o : objSAtts\ (\theta ActualPermSet, u?, \{Read, ReadWrite\})$
$\bullet \neg\ objclass\ o?\ dominates\ objclass\ o)$

$GetWriteEObjNotExist \mathrel{\widehat=} ErrorObjDoesNotExist[function := fGetWrite]$

$GetWriteEInvalidUser \mathrel{\widehat=} ErrorInvalidUser[function := fGetWrite]$

$GetWriteNotAffected \mathrel{\widehat=} \Xi UsrObj \land \Xi SecClasses \land \Xi ACLS$

$GetWriteReject \mathrel{\widehat=} \Xi ActualPermSet \land (GetWriteOkNo \lor GetWriteEInvalidUser \lor GetWriteEObjNotExist)$

$GetWrite \mathrel{\widehat=} GetWriteNotAffected \land (GetWriteOkYes \lor GetWriteReject)$

GetExecuteOkYes

$AugmentingAccess[u?/u, o?/o, a := Execute]$
$ACLS$
$out! : DECISION$

$o? \in oids \land u? \in uids$
$out! = YES$
$(u?, Execute) \in acl\ o?$

GetExecuteOkNo

$$
\begin{array}{l}
ACLS \\
u? : UID \\
o? : OID \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \\
(u?, Execute) \notin acl\ o? \\
out! = NO
\end{array}
$$

$$GetExecuteInvalidUser \mathrel{\widehat=} ErrorInvalidUser[function := fGetExecute]$$

$$GetExecuteEObjNotExist \mathrel{\widehat=} ErrorObjDoesNotExist[function := fGetExecute]$$

$$GetExecuteENotAffected \mathrel{\widehat=} \Xi UsrObj \land \Xi SecClasses \land \Xi ACLS$$

$$GetExecuteReject \mathrel{\widehat=} \Xi ActualPermSet \land (GetExecuteOkNo \lor GetExecuteInvalidUser \lor GetExecuteEObjNotExist)$$

$$GetExecute \mathrel{\widehat=} GetExecuteENotAffected \land (GetExecuteOkYes \lor GetExecuteReject)$$

GetReadWriteOkYes

$$
\begin{array}{l}
AugmentingAccess[u?/u, o?/o, a := ReadWrite] \\
SecClasses \\
ACLS \\
out! : DECISION \\
\hline
o? \in oids \land u? \in uids \\
(u?, ReadWrite) \in acl\ o? \\
usrclass\ u?\ dominates\ objclass\ o? \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Read\}) \bullet \\
\quad objclass\ o?\ dominates\ objclass\ o \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{Write\}) \bullet \\
\quad objclass\ o\ dominates\ objclass\ o? \\
\forall o : objSAtts\ (\theta ActualPermSet, u?, \{ReadWrite\}) \bullet \\
\quad objclass\ o?\ matches\ objclass\ o \\
out! = YES
\end{array}
$$

GetReadWriteOkNo

$ActualPermSet$
$u? : UID$
$o? : OID$
$SecClasses$
$ACLS$
$out! : DECISION$

$o? \in oids \land u? \in uids$
$\neg\ ((u?, ReadWrite) \in acl\ o?$
$\quad \land\ usrclass\ u?\ dominates\ objclass\ o?$
$\quad \land\ (\forall\ o : objSAtts\ (\theta ActualPermSet, u?, \{Read\}) \bullet$
$\qquad objclass\ o?\ dominates\ objclass\ o)$
$\quad \land\ (\forall\ o : objSAtts\ (\theta ActualPermSet, u?, \{Write\}) \bullet$
$\qquad objclass\ o\ dominates\ objclass\ o?)$
$\quad \land\ (\forall\ o : objSAtts\ (\theta ActualPermSet, u?, \{ReadWrite\}) \bullet$
$\qquad objclass\ o?\ matches\ objclass\ o))$
$out! = NO$

$$GetReadWriteEObjNotExist \mathrel{\widehat=} ErrorObjDoesNotExist[function := fGetReadWrite]$$

$$GetReadWriteEInvalidUser \mathrel{\widehat=} ErrorInvalidUser[function := fGetReadWrite]$$

$$GetReadWriteNotAffected \mathrel{\widehat=} \Xi UsrObj \land \Xi SecClasses \land \Xi ACLS$$

$$GetReadWriteReject \mathrel{\widehat=} \Xi ActualPermSet \land (GetReadWriteOkNo \lor GetReadWriteEInvalidUser \lor GetReadWriteEObjNotExist)$$

$$GetReadWrite \mathrel{\widehat=} GetReadWriteNotAffected \land (GetReadWriteOkYes \lor GetReadWriteReject)$$

ReleaseOk

$\Delta ActualPermSet$
$u? : UID$
$o? : OID$
$a? : PERMISSION$
$out! : DECISION$

$o? \in oids$
$u? \in uids$
$permSet' = dimb\ (permSet, (u?, o?), a?)$
$out! = YES$

$$ReleaseEObjNotExist \mathrel{\widehat{=}} ErrorObjDoesNotExist[function := fRelease]$$

$$ReleaseEInvalidUser \mathrel{\widehat{=}} ErrorInvalidUser[function := fRelease]$$

$$ReleaseNotAffected \mathrel{\widehat{=}} \Xi UsrObj \land \Xi SecClasses \land \Xi ACLS$$

$$ReleaseReject \mathrel{\widehat{=}} \Xi ActualPermSet \land (ReleaseEInvalidUser \lor ReleaseEObjNotExist)$$

$$Release \mathrel{\widehat{=}} ReleaseNotAffected \land (ReleaseOk \lor ReleaseReject)$$

GiveOkYes

$$\Delta ACLS$$
$$u? : UID$$
$$o? : OID$$
$$a? : PERMISSION$$
$$target? : UID$$
$$out! : DECISION$$

$$o? \in oids \land u? \in uids \land target? \in uids$$
$$out! = YES$$
$$(u?, a?) \in acl\ o?$$
$$(u?, Control) \in acl\ o?$$
$$acl' = acl \oplus \{(o? \mapsto acl\ o? \cup \{(target?, a?)\})\}$$

GiveOkNo

$$ACLS$$
$$u? : UID$$
$$target? : UID$$
$$o? : OID$$
$$a? : PERMISSION$$
$$out! : DECISION$$

$$o? \in oids \land u? \in uids$$
$$\neg (target? \in uids \land (u?, a?) \in acl\ o? \land (u?, Control) \in acl\ o?)$$
$$out! = NO$$

$$GiveEObjNotExist \mathrel{\widehat{=}} ErrorObjDoesNotExist[function := fGive]$$

$$GiveEInvalidUser \mathrel{\widehat{=}} ErrorInvalidUser[function := fGive]$$

$$GiveNotAffected \mathrel{\widehat{=}} \Xi UsrObj \land \Xi SecClasses \land \Xi ActualPermSet$$

$$GiveReject \mathrel{\widehat=} \Xi ACLS \land (GiveOkNo \lor GiveEInvalidUser \lor GiveEObjNotExist)$$

$$Give \mathrel{\widehat=} GiveNotAffected \land (GiveOkYes \lor GiveReject)$$

RescindOkYes

$\Delta ACLS$
$u? : UID$
$o? : OID$
$a? : PERMISSION$
$target? : UID$
$out! : DECISION$

$o? \in oids \land u? \in uids \land target? \in uids$
$out! = YES$
$(u?, a?) \in acl\ o?$
$(u?, Control) \in acl\ o?$
$acl' = acl \oplus \{(o? \mapsto acl\ o? \setminus \{(target?, a?)\})\}$

RescindOkNo

$ACLS$
$u? : UID$
$o? : OID$
$a? : PERMISSION$
$target? : UID$
$out! : DECISION$

$o? \in oids \land u? \in uids$
$\neg\ (target? \in uids \land (u?, a?) \in acl\ o? \land (u?, Control) \in acl\ o?)$
$out! = NO$

$$RescindEObjNotExist \mathrel{\widehat=} ErrorObjDoesNotExist[function := fRescind]$$

$$RescindEInvalidUser \mathrel{\widehat=} ErrorInvalidUser[function := fRescind]$$

$$RescindNotAffected \mathrel{\widehat=} \Xi UsrObj \land \Xi SecClasses \land \Xi ActualPermSet$$

$$RescindReject \mathrel{\widehat=} \Xi ACLS \land (RescindOkNo \lor RescindEInvalidUser \lor RescindEObjNotExist)$$

$$Rescind \mathrel{\widehat=} RescindNotAffected \land (RescindOkYes \lor RescindReject)$$

ChangeObjClassOkYes

$ActualPermSet$
$MACAdmin$
$\Delta SecClasses$
$u? : UID$
$o? : OID$
$secClass? : SecClass$
$out! : DECISION$

$o? \in oids \land u? \in uids \land u? \in macAdmins$
$\forall u : uids \bullet permSet\ (u, o?) = \emptyset$
$out! = YES$
$objclass' = objclass \oplus \{(o? \mapsto secClass?)\}$
$usrclass' = usrclass$

ChangeObjClassOkNo

$ActualPermSet$
$MACAdmin$
$\Delta SecClasses$
$u? : UID$
$o? : OID$
$out! : DECISION$

$o? \in oids \land u? \in uids$
$\exists u : uids \bullet permSet\ (u, o?) \neq \emptyset \lor u? \notin macAdmins$
$out! = NO$

$$ChangeObjClassNotAffected \mathrel{\widehat=} \Xi UsrObj \land \Xi ActualPermSet \land \Xi ACLS$$

$$ChangeObjClassEInvalidUser \mathrel{\widehat=} ErrorInvalidUser\ [function := fChangeObjClass]$$

$$ChangeObjClassEObjNotExist \mathrel{\widehat=} ErrorObjDoesNotExist\ [function := fChangeObjClass]$$

$$ChangeObjClassReject \mathrel{\widehat=} \Xi SecClasses \land (ChangeObjClassOkNo \lor ChangeObjClassEInvalidUser \lor ChangeObjClassEObjNotExist)$$

$$ChangeObjClass \mathrel{\widehat=} ChangeObjClassNotAffected \land (ChangeObjClassOkYes \lor ChangeObjClassReject)$$

CreateObjectOkYes

$\Delta ACLS$
$\Delta ActualPermSet$
$\Delta SecClasses$
$MACAdmin$
$u? : UID$
$o? : OID$
$out! : DECISION$

$u? \in uids \land o? \notin oids \land u? \in macAdmins$
$out! = YES$
$oids' = oids \cup \{o?\} \land uids' = uids$
$acl' = acl \oplus \{(o? \mapsto \emptyset)\}$
$objclass' = objclass \oplus \{(o? \mapsto InitOSecClass)\}$
$usrclass' = usrclass$
$permSet' = permSet \oplus \{u : uids \bullet ((u, o?) \mapsto \emptyset)\}$

CreateObjectOkNo

$\Xi ACLS$
$\Xi ActualPermSet$
$\Xi SecClasses$
$MACAdmin$
$u? : UID$
$o? : OID$
$out! : DECISION$

$u? \in uids \land o? \notin oids$
$u? \notin macAdmins$
$out! = NO$

CreateObjectEInvalidUser

$Error[InvalidUser/message, function := fCreateObject]$
$u? : UID$

$u? \notin uids$

CreateObjectEObjAlreadyExists

$Error[ObjAlreadyExists/message, function := fCreateObject]$
$UsrObj$
$o? : OID$

$o? \in oids$

$$CreateObjectReject \mathrel{\widehat=} \Xi SecClasses \land \Xi ACLS \land \Xi ActualPermSet \land (CreateObjectOkNo \lor CreateObjectEInvalidUser \lor CreateObjectEObjAlreadyExists)$$

$$CreateObject \mathrel{\widehat=} CreateObjectOkYes \lor CreateObjectReject$$

DeleteObjectOkYes

$$\Delta ACLS$$
$$\Delta ActualPermSet$$
$$\Delta SecClasses$$
$$MACAdmin$$
$$u? : UID$$
$$o? : OID$$
$$out! : DECISION$$

$$u? \in uids \land o? \in oids \land u? \in macAdmins$$
$$oids' = oids \setminus \{o?\} \land uids' = uids$$
$$permSet' = \{u : uids \bullet (u, o?)\} \lhd permSet$$
$$acl' = \{o?\} \lhd acl$$
$$usrclass' = usrclass$$
$$objclass' = \{o?\} \lhd objclass$$
$$out! = YES$$

DeleteObjectOkNo

$$\Delta ACLS$$
$$\Delta ActualPermSet$$
$$\Delta SecClasses$$
$$MACAdmin$$
$$u? : UID$$
$$o? : OID$$
$$out! : DECISION$$

$$u? \in uids \land o? \in oids$$
$$u? \notin macAdmins$$
$$out! = NO$$

$$DeleteObjectEObjNotExist \mathrel{\widehat=} ErrorObjDoesNotExist[function := fDeleteObject]$$

DeleteObjectEInvalidUser

$$Error[InvalidUser/message, function := fDeleteObject]$$
$$u? : UID$$

$$u? \notin uids$$

$$DeleteObjectReject \mathrel{\widehat=} \Xi SecClasses \land \Xi ACLS \land \Xi ActualPermSet \land (DeleteObjectOkNo \lor DeleteObjectEInvalidUser \lor DeleteObjectEObjNotExist)$$

$$DeleteObject \mathrel{\widehat=} DeleteObjectOkYes \lor DeleteObjectReject$$

Apéndice C

Pruebas completas para Z/EVES 2.1

Las siguientes constituyen todas las pruebas realizadas para la especificación en Z/EVES. Están listadas en orden de prueba, es decir, las cada una puede sólo los teoremas del asistente de pruebas y las anteriormente probadas. Fueron desarrolladas usando la versión de Z/EVES 2.1.

Theorem rule math_funElemInImg[X, Y]
$\forall f : X \leftrightarrow Y \bullet (x, y) \in f \Rightarrow y \in Y$

proof)
invoke $\_ \leftrightarrow \_$
apply *inPower* to predicate
$f \in \mathbb{P}(X \times Y)$
instantiate $e == (x, y)$
rearrange
with predicate $((x, y) \in f)$
simplify
apply *tupleInCross2* to predicate
$(x, y) \in X \times Y$
simplify
∎

Theorem rule math_funElemInDom[X, Y]
$\forall f : X \leftrightarrow Y \bullet (x, y) \in f \Rightarrow x \in X$

proof)
invoke $\_ \leftrightarrow \_$
apply *inPower* to predicate
$f \in \mathbb{P}(X \times Y)$
instantiate $e == (x, y)$
rearrange

prove
■

Theorem rule math_funIsRel[X, Y]
$f \in X \pfun Y \Rightarrow f \in X \rel Y$

proof)
prove
■

Theorem rule math_eqApp
$x = y \Rightarrow f\ x = f\ y$

proof)
prove
■

Theorem rule mkerrType
$mkerr \in MESSAGE \times FUNCTION \rel DECISION$

proof)
use *mkerr$declaration*
invoke $\_ \fun \_$
invoke $(\_ \pfun \_)$
prove by reduce
■

Theorem rule PERMS_in_RIGHTS
$a \in \power PERMISSION \Rightarrow a \in \power RIGHT$

proof)
prove by reduce
■

Theorem grule gDATT_in_ATT
$\forall a : \power PERMISSION \bullet a \in \power RIGHT$

proof)
with enabled (*PERMS_in_RIGHTS*)
prove
■

Theorem oplus_implies[X, Y]
$\forall f : X \pfun Y;\ g : X \pfun Y;\ h : X \pfun Y \bullet$
$h = f \oplus g \Rightarrow (\dom h = \dom f \cup \dom g) \land$
$(\forall x : \dom f \bullet (x \notin \dom g \Rightarrow h\ x = f\ x)) \land$
$(\forall y : \dom g \bullet h\ y = g\ y)$

proof)
prove by reduce
∎

Theorem rule fUO_PERMS_in_fUO_RIGHT
$f \in UID \times OID \nrightarrow \mathbb{P}\, PERMISSION$
$\Rightarrow f \in UID \times OID \nrightarrow \mathbb{P}\, RIGHT$

proof)
prove by reduce
∎

Theorem rule UOp_str
$p \in UID \times OID \Leftrightarrow p \in UOp$

proof)
prove by reduce
∎

Theorem oplus_maintainDom
$\forall f : fUO\_PERMS;\ p : UOp;\ a : \mathbb{P}\, PERMISSION \mid$
$p \in \mathrm{dom}\, f \bullet \mathrm{dom}\, f = \mathrm{dom}(f \oplus \{(p \mapsto a)\})$
proof)
apply *extensionality* to predicate
$\mathrm{dom}\, f = \mathrm{dom}(f \oplus \{(p \mapsto a)\})$
cases
prenex
prove by reduce
next
prenex
prove by reduce
split $y = p$
cases
prove
next
prove
next
∎

Theorem augb$DomainCheck

proof)
cases
prove by reduce
next
prove by reduce
next
∎

Theorem rule fUOPERMS_sss
$f \in fUO\_PERMS \Rightarrow f \in UOp \pfun \mathbb{P}\, PERMISSION$

proof)
prove by reduce
∎

Theorem rule fUOPERMS_gss
$f \in fUO\_PERMS \Rightarrow f \in UID \times OID \pfun \mathbb{P}\, PERMISSION$

proof)
prove by reduce
∎

Theorem rule fUOPERMS_gsg
$f \in fUO\_PERMS \Rightarrow f \in UID \times OID \pfun \mathbb{P}\, RIGHT$

proof)
prove by reduce
∎

Theorem rule fUOPERMS_sgs
$f \in fUO\_PERMS \Rightarrow f \in UOp \rel \mathbb{P}\, PERMISSION$

proof)
prove by reduce
∎

Theorem rule fUOPERMS_ggg
$f \in fUO\_PERMS \Rightarrow f \in UID \times OID \rel \mathbb{P}\, RIGHT$

proof)
prove by reduce
∎

Theorem rule fUOPERMS_dom
$\forall f : fUO\_PERMS \bullet \mathrm{dom}\, f = \mathrm{dom}[UOp, (\mathbb{P}\, PERMISSION)]\, f$

proof)
apply *extensionality* to predicate
$\mathrm{dom}\, f = \mathrm{dom}[UOp, (\mathbb{P}\, PERMISSION)]\, f$
cases
prenex
apply *inDom* to predicate
$x \in \mathrm{dom}[UOp, (\mathbb{P}\, PERMISSION)]\, f$
apply *inDom* to predicate
$x \in \mathrm{dom}\, f$

apply *fUOPERMS_ggg* to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
apply *fUOPERMS_sgs* to predicate
$f \in UOp \leftrightarrow \mathbb{P}\, PERMISSION$
simplify
prenex
instantiate $y\_0 == y$
use *math_funElemInImg*$[UOp, \mathbb{P}\, PERMISSION]$
rearrange
invoke *fUO_PERMS*
prove
next
prenex
apply *inDom* to predicate
$y \in \mathrm{dom}\, f$
apply *inDom* to predicate
$y \in \mathrm{dom}[UOp, (\mathbb{P}\, PERMISSION)]\ f$
apply *fUOPERMS_sgs* to predicate
$f \in UOp \leftrightarrow \mathbb{P}\, PERMISSION$
apply *fUOPERMS_ggg* to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
simplify
prenex
instantiate $y\_1 == y\_0$
apply *PERMS_in_RIGHTS* to predicate
$y \in \mathbb{P}\, RIGHT$
prove
next
∎

Theorem rule fUOPERMS_DomApp
$\forall f : fUO\_PERMS;\ p : UOp \mid p \in \mathrm{dom}\, f \bullet f\ p \in \mathbb{P}\, PERMISSION$

proof)
with enabled (*fUOPERMS_dom*)
prove by reduce
∎

Theorem rule augb_implicationsHelperLemma
$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom}\, f \bullet f\ up \cup \{na\} \in \mathbb{P}\, PERMISSION$
proof)
simplify
apply *cupSubset* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\, PERMISSION$
apply *PERMS_in_RIGHTS* to predicate
$\{na\} \in \mathbb{P}\, RIGHT$
apply *unitSubset* to predicate
$\{na\} \in \mathbb{P}\, PERMISSION$
simplify

apply *PERMS_in_RIGHTS* to predicate
$f\ up \in \mathbb{P}\ RIGHT$
apply *fUOPERMS_DomApp* to predicate
$f\ up \in \mathbb{P}\ PERMISSION$
simplify
apply *inDom* to predicate
$up \in \mathrm{dom}[UOp, (\mathbb{P}\ PERMISSION)]\ f$
■

Theorem rule augb_implicationsHelperLemma_bis
$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom}\, f \bullet f\ up \cup \{na\} \in \mathbb{P}\ RIGHT$
proof)
simplify
apply *PERMS_in_RIGHTS* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\ RIGHT$
apply *augb_implicationsHelperLemma* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\ PERMISSION$
simplify
■

Theorem rule augb_implicationsHelperLemma5
$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom}\, f \bullet up \mapsto f\ up \cup \{na\} \in (UID \times OID) \times \mathbb{P}\ RIGHT$
proof)
apply *mapDef* to expression
$up \mapsto f\ up \cup \{na\}$
apply *UOp_str* to predicate
$up \in UID \times OID$
apply *augb_implicationsHelperLemma_bis* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\ RIGHT$
simplify
with predicate $((up, f\ up \cup \{na\}) \in (UID \times OID) \times \mathbb{P}\ RIGHT)$
rewrite
apply *UOp_str* to predicate
$up \in UID \times OID$
simplify
■

Theorem rule augb_implicationsHelperLemma2
$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom}\, f \bullet \{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \nrightarrow \mathbb{P}\ RIGHT$
proof)
simplify
apply *unitInPfun* to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \nrightarrow \mathbb{P}\ RIGHT$
apply *augb_implicationsHelperLemma5* to predicate
$up \mapsto f\ up \cup \{na\} \in (UID \times OID) \times \mathbb{P}\ RIGHT$

simplify
∎

Theorem rule augb_implicationsHelperLemma3
$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom} f \bullet \{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
proof)
prove by reduce
∎

Theorem rule augb_app
$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom} f \bullet (f \oplus \{(up \mapsto f\ up \cup \{na\})\})up = f\ up \cup \{na\}$
proof)
apply *applyOverride* to expression
$(f \oplus \{(up \mapsto f\ up \cup \{na\})\})up$
apply *augb_implicationsHelperLemma3* to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
apply *pfunAppliesTo* to predicate
$(\{up \mapsto f\ up \cup \{na\}\}, up) \in applies\$to$
apply *augb_implicationsHelperLemma2* to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \nrightarrow \mathbb{P}\, RIGHT$
apply *fUOPERMS_ggg* to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
simplify
apply *domSingleton* to expression
$\mathrm{dom}\{(up \mapsto f\ up \cup \{na\})\}$
apply *mapDef* to expression
$up \mapsto f\ up \cup \{na\}$
apply *UOp_str* to predicate
$up \in UID \times OID$
apply *augb_implicationsHelperLemma_bis* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\, RIGHT$
simplify
apply *tupleInCross2* to predicate
$(up, f\ up \cup \{na\}) \in (UID \times OID) \times \mathbb{P}\, RIGHT$
apply *UOp_str* to predicate
$up \in UID \times OID$
apply *augb_implicationsHelperLemma_bis* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\, RIGHT$
simplify
rewrite
∎

Theorem rule augb_app2
$\forall f : fUO\_PERMS;\ up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom} f \bullet augb\ (f, up, na)\ up = f\ up \cup \{na\}$
proof)

apply *augb_def* to expression
augb (f, up, na)
with enabled (*augb_app*)
prove
∎

Theorem rule augb_domFEqu
$\forall g : fUO\_PERMS;\ f : fUO\_PERMS;$
$up : UOp;\ na : PERMISSION \mid$
$up \in \operatorname{dom} f \land g = augb\ (f, up, na) \bullet \operatorname{dom} g = \operatorname{dom} f$
proof)
apply *augb_def* to expression
augb (f, up, na)
simplify
equality substitute
apply *domOverride* to expression
$\operatorname{dom}(f \oplus \{(up \mapsto f\ up \cup \{na\})\})$
apply *fUOPERMS_ggg* to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
apply *augb_implicationsHelperLemma3* to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
simplify
apply *cupSubsetRight* to expression
$\operatorname{dom} f \cup \operatorname{dom}\{(up \mapsto f\ up \cup \{na\})\}$
apply *domSingleton* to expression
$\operatorname{dom}\{(up \mapsto f\ up \cup \{na\})\}$
apply *mapDef* to expression
$up \mapsto f\ up \cup \{na\}$
apply *UOp_str* to predicate
$up \in UID \times OID$
apply *augb_implicationsHelperLemma_bis* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\, RIGHT$
simplify
apply *tupleInCross2* to predicate
$(up, f\ up \cup \{na\}) \in (UID \times OID) \times \mathbb{P}\, RIGHT$
apply *UOp_str* to predicate
$up \in UID \times OID$
apply *augb_implicationsHelperLemma_bis* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\, RIGHT$
simplify
apply *subsetDef* to predicate
$\{up\} \subseteq \operatorname{dom} f$
apply *domInPower* to predicate
$\operatorname{dom} f \in \mathbb{P}(UID \times OID)$
apply *inPowerSelf* to predicate
$UID \times OID \in \mathbb{P}(UID \times OID)$
apply *fUOPERMS_ggg* to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
simplify
prove
∎

Theorem rule augb_aGrow
$\forall g : fUO\_PERMS;\ f : fUO\_PERMS;$
$up : UOp;\ na : PERMISSION;\ op : UOp \mid$
$up \in \mathrm{dom}\, f \land g = augb\ (f, up, na) \land op \in \mathrm{dom}\, f \bullet$
$f\ op \subseteq g\ op$
proof)
apply *augb_def* to expression
$augb\ (f, up, na)$
simplify
equality substitute
split $op = up$
cases
apply *applyOverride* to expression
$(f \oplus \{(up \mapsto f\ up \cup \{na\})\})op$
apply *augb_implicationsHelperLemma3* to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
apply *fUOPERMS_ggg* to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
apply *UOp_str* to predicate
$op \in UID \times OID$
apply *pfunAppliesTo* to predicate
$(\{up \mapsto f\ up \cup \{na\}\}, op) \in applies\$to$
apply *augb_implicationsHelperLemma2* to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \nrightarrow \mathbb{P}\, RIGHT$
apply *domSingleton* to expression
$\mathrm{dom}\{(up \mapsto f\ up \cup \{na\})\}$
apply *augb_implicationsHelperLemma5* to predicate
$up \mapsto f\ up \cup \{na\} \in (UID \times OID) \times \mathbb{P}\, RIGHT$
simplify
apply *mapDef* to expression
$up \mapsto f\ up \cup \{na\}$
apply *UOp_str* to predicate
$up \in UID \times OID$
apply *augb_implicationsHelperLemma_bis* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\, RIGHT$
simplify
apply *inUnit* to predicate
$op \in \{up\}$
simplify
apply *applyUnit* to expression
$\{(up, (f\ up \cup \{na\}))\}\ op$
simplify
apply *subDef* to predicate
$f\ op \subseteq f\ up \cup \{na\}$
apply *PERMS_in_RIGHTS* to predicate
$f\ op \in \mathbb{P}\, RIGHT$
apply *fUOPERMS_DomApp* to predicate
$f\ op \in \mathbb{P}\, PERMISSION$
apply *augb_implicationsHelperLemma_bis* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\, RIGHT$
simplify
prenex

apply *inCup* to predicate
$x \in f\ up \cup \{na\}$
apply *PERMS_in_RIGHTS* to predicate
$f\ up \in \mathbb{P}\, RIGHT$
apply *fUOPERMS_DomApp* to predicate
$f\ up \in \mathbb{P}\, PERMISSION$
apply *PERMS_in_RIGHTS* to predicate
$\{na\} \in \mathbb{P}\, RIGHT$
apply *unitSubset* to predicate
$\{na\} \in \mathbb{P}\, PERMISSION$
simplify
next
apply *applyOverride*2 to expression
$(f \oplus \{(up \mapsto f\ up \cup \{na\})\})\ op$
apply *fUOPERMS_ggg* to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
apply *augb_implicationsHelperLemma*3 to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
apply *UOp_str* to predicate
$op \in UID \times OID$
apply *domSingleton* to expression
$\mathrm{dom}\{(up \mapsto f\ up \cup \{na\})\}$
apply *augb_implicationsHelperLemma*5 to predicate
$up \mapsto f\ up \cup \{na\} \in (UID \times OID) \times \mathbb{P}\, RIGHT$
apply *pfunAppliesTo* to predicate
$(f, op) \in applies\$to$
apply *fUOPERMS_gsg* to predicate
$f \in UID \times OID \nrightarrow \mathbb{P}\, RIGHT$
simplify
apply *inUnit* to predicate
$op \in \{(up \mapsto f\ up \cup \{na\}).1\}$
apply *mapDef* to expression
$up \mapsto f\ up \cup \{na\}$
apply *UOp_str* to predicate
$up \in UID \times OID$
apply *augb_implicationsHelperLemma_bis* to predicate
$f\ up \cup \{na\} \in \mathbb{P}\, RIGHT$
simplify
apply *subsetSelf* to predicate
$f\ op \subseteq f\ op$
with predicate $(\neg\ f\ op \in \mathbb{P}\, RIGHT)$
rewrite
apply *PERMS_in_RIGHTS* to predicate
$f\ op \in \mathbb{P}\, RIGHT$
apply *fUOPERMS_DomApp* to predicate
$f\ op \in \mathbb{P}\, PERMISSION$
simplify
next
■

Theorem rule augb_naEqu
$\forall g : fUO\_PERMS;\ f : fUO\_PERMS;$
$up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom} f \land g = augb\ (f, up, na) \bullet$
$\forall p : \mathrm{dom} f \mid p \neq up \bullet g\ p = f\ p$
proof)
apply *augb_def* to expression
augb (f, up, na)
prove by reduce
∎

Theorem rule augb_aNoDim
$\forall g : fUO\_PERMS;\ f : fUO\_PERMS;$
$up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom} f \land g = augb\ (f, up, na)$
$\bullet \forall p : \mathrm{dom} f \bullet \forall a : g\ p \mid na \neq a \bullet a \in f\ p$
proof)
apply *augb_def* to expression
augb (f, up, na)
simplify
equality substitute
split $p = up$
cases
apply *applyOverride* to expression
$(f \oplus \{(up \mapsto f\ up \cup \{na\})\})\ p$
apply *fUOPERMS_ggg* to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\ RIGHT$
apply *augb_implicationsHelperLemma3* to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \leftrightarrow \mathbb{P}\ RIGHT$
simplify
apply *pfunAppliesTo* to predicate
$(\{up \mapsto f\ up \cup \{na\}\}, p) \in applies\$to$
apply *augb_implicationsHelperLemma2* to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \nrightarrow \mathbb{P}\ RIGHT$
simplify
apply *domSingleton* to expression
$\mathrm{dom}\{(up \mapsto f\ up \cup \{na\})\}$
apply *augb_implicationsHelperLemma5* to predicate
$up \mapsto f\ up \cup \{na\} \in (UID \times OID) \times \mathbb{P}\ RIGHT$
with expression $(\{(up \mapsto f\ up \cup \{na\}).1\})$
rewrite
simplify
apply *inUnit* to predicate
$p \in \{up\}$
simplify
rearrange
equality substitute
with expression $(\{(up \mapsto f\ up \cup \{na\})\}\ p)$
rewrite
apply *inCup* to predicate
$a \in f\ up \cup \{na\}$

apply $PERMS\_in\_RIGHTS$ to predicate
$f\ up \in \mathbb{P}\, RIGHT$
apply $fUOPERMS\_DomApp$ to predicate
$f\ up \in \mathbb{P}\, PERMISSION$
apply $PERMS\_in\_RIGHTS$ to predicate
$\{na\} \in \mathbb{P}\, RIGHT$
apply $unitSubset$ to predicate
$\{na\} \in \mathbb{P}\, PERMISSION$
simplify
apply $inUnit$ to predicate
$a \in \{na\}$
apply $notEqRule$ to predicate
$na \neq a$
simplify
apply UOp_str to predicate
$p \in UID \times OID$
simplify
next
apply $applyOverride2$ to expression
$(f \oplus \{(up \mapsto f\ up \cup \{na\})\})\ p$
apply $domSingleton$ to expression
$\mathrm{dom}\{(up \mapsto f\ up \cup \{na\})\}$
apply $augb\_implicationsHelperLemma5$ to predicate
$up \mapsto f\ up \cup \{na\} \in (UID \times OID) \times \mathbb{P}\, RIGHT$
apply $augb\_implicationsHelperLemma3$ to predicate
$\{up \mapsto f\ up \cup \{na\}\} \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
apply $fUOPERMS\_ggg$ to predicate
$f \in UID \times OID \leftrightarrow \mathbb{P}\, RIGHT$
simplify
apply $pfunAppliesTo$ to predicate
$(f, p) \in applies\$to$
apply UOp_str to predicate
$p \in UID \times OID$
apply $fUO\_PERMS\_in\_fUO\_RIGHT$ to predicate
$f \in UID \times OID \nrightarrow \mathbb{P}\, RIGHT$
apply $fUOPERMS\_gss$ to predicate
$f \in UID \times OID \nrightarrow \mathbb{P}\, PERMISSION$
simplify
next
prove by reduce
■

Theorem rule augb_naNoDim
$\forall g : fUO\_PERMS;\ f : fUO\_PERMS;$
$up : UOp;\ na : PERMISSION \mid$
$up \in \mathrm{dom}\, f \land g = augb\ (f, up, na) \bullet$
$\forall p : \mathrm{dom}\, f \bullet \forall a : g\ p \mid (up, na) \neq (p, a) \bullet a \in f\ p$
proof)
use $augb\_naEqu$
use $augb\_aNoDim$
rearrange

simplify
with predicate $((up, na) \neq (p, a))$
rewrite
split $up = p$
cases
prove
next
prove
next
■

Theorem dimb$DomainCheck

proof)
prove by reduce
■

Theorem secureRel$DomainCheck

proof)
apply *SecClasses$member* to predicate
$f \in SecClasses$
prenex
apply *pfunAppliesTo* to predicate
$(f.usrclass, u) \in applies\$to$
apply *pfunAppliesTo* to predicate
$(f.objclass, o) \in applies\$to$
invoke *SecClasses*
rearrange
invoke *fUC*
invoke *fOC*
rewrite
■

Theorem rule permissionSet_dom
$ActualPermSet \wedge u \in uids \wedge o \in oids \Rightarrow (u, o) \in \mathrm{dom}\ permSet$

proof)
invoke *ActualPermSet*
apply *extensionality* to predicate
$\mathrm{dom}\ permSet = uids \times oids$
instantiate $y == (u, o)$
prove
■

Theorem rule permissionSet_domp
$ActualPermSet \wedge p \in uids \times oids \Rightarrow p \in \mathrm{dom}\ permSet$

proof)

invoke *ActualPermSet*
prove
∎

Theorem rule permissionSet_domUID
$ActualPermSet \land (u, o) \in \text{dom}\, permSet \Rightarrow u \in uids$

proof)
prove by reduce
invoke *ActualPermSet*
apply *extensionality* to predicate
$\text{dom}\, permSet = uids \times oids$
instantiate $x == (u, o)$
prove
∎

Theorem rule permissionSet_domOID
$ActualPermSet \land (u, o) \in \text{dom}\, permSet \Rightarrow o \in oids$

proof)
invoke *ActualPermSet*
apply *extensionality* to predicate
$\text{dom}\, permSet = uids \times oids$
instantiate $x == (u, o)$
prove
∎

Theorem objSAtts$DomainCheck

proof)
prenex
cases
apply *ActualPermSet$member* to predicate
$m \in ActualPermSet$
prenex
apply *extensionality* to predicate
$\text{dom}\, \textbf{local}\ objSAtts =$
$\{atts\_0 : \mathbb{P}\, PERMISSION \mid u\_0 \in m\_0.uids\}$
instantiate $y == (m, u, atts)$
with predicate $((m, u, atts) \in$
$\{atts\_1 : \mathbb{P}\, PERMISSION \mid u\_1 \in m\_1.uids\})$
rewrite
rearrange
apply *ActualPermSet$member* to predicate
$m \in ActualPermSet$
prove
next
apply *pfunAppliesTo* to predicate
$(m\_0.permSet, (u\_0, o)) \in applies\$to$

apply *ActualPermSet$member* to predicate
m__0 ∈ *ActualPermSet*
prenex
invoke *ActualPermSet*
cases
apply *extensionality* to predicate
dom **local** *objSAtts* =
{*atts__1* : ℙ *PERMISSION* | *u__1* ∈ *m__1.uids*}
instantiate *y* == (*θActualPermSet*, *u__0*, *atts__0*)
rearrange
with predicate ((*θActualPermSet*, *u*, *atts*) ∈
{*atts__3* : ℙ *PERMISSION* | *u__3* ∈ *m__3.uids*})
rewrite
apply *ActualPermSet$thetaMember* to predicate
θActualPermSet ∈ *ActualPermSet*
invoke *ActualPermSet*
with predicate (*UsrObj* ∧ *permSet* ∈ *fUO_PERMS* ∧
dom *permSet* = *uids* × *oids*)
simplify
rewrite
next
apply *fUOPERMS_gsg* to predicate
m__0.permSet ∈ *UID* × *OID* ⇸ ℙ *RIGHT*
simplify
apply *extensionality* to predicate
dom *permSet* = *uids* × *oids*
instantiate *y* == (*u__0*, *o*)
apply *tupleInCross2* to predicate
(*u*, *o*) ∈ *uids* × *oids*
rearrange
simplify
next
∎

Theorem rule objSAtts_ran
ActualPermSet ∧ *u* ∈ *uids* ∧ *pa* ∈ ℙ *PERMISSION*
⇒ *objSAtts*(*θActualPermSet*, *u*, *pa*) ∈ ℙ *OID*

proof)
apply *objSAtts_def*1 to expression
objSAtts (*θActualPermSet*, *u*, *pa*)
prove
∎

Theorem SimpleSecureState$DomainCheck

proof)
invoke *ActualPermSet*
prove by reduce
∎

Theorem rule SimpleSecurityNotAffected
$\Xi ActualPermSet \land \Xi SecClasses \Rightarrow PreservesSimpleSecurity$

proof)
invoke *PreservesSimpleSecurity*
invoke $\Xi ActualPermSet$
invoke $\Xi SecClasses$
invoke $\Delta ActualPermSet$
invoke $\Delta SecClasses$
simplify
■

Theorem StarPropertyState$DomainCheck

proof)
apply *objSAtts_dom* to expression
dom *objSAtts*
prove by reduce
apply *objSAtts_def* 1 to expression
$objSAtts(\theta ActualPermSet[oids := \text{dom } objclass, uids := \text{dom } usrclass], u, (\{Write\} \cup \{ReadWrite\}))$
apply *objSAtts_def* 1 to expression
$objSAtts(\theta ActualPermSet[oids := \text{dom } objclass, uids := \text{dom } usrclass], u, (\{Read\} \cup \{ReadWrite\}))$
prove by reduce
■

Theorem rule StarPropertyNotAffected
$\Xi ActualPermSet \land \Xi SecClasses \Rightarrow PreservesStarProperty$

proof)
invoke *PreservesStarProperty*
invoke $\Delta ActualPermSet$
invoke $\Delta SecClasses$
invoke *StarPropertyState*
simplify
prenex
prove by reduce
instantiate $u\_0 == u$
rearrange
simplify
instantiate $ow\_1 == ow, or\_1 == or$
simplify
■

Theorem AugmentingAccess$DomainCheck

proof)

apply *augb_dom* to expression
dom *augb*
with disabled (*fUOPERMS_dom*)
rewrite
invoke *UOp*
apply *tupleInCross2* to predicate
$(u, o) \in UID \times OID$
simplify
∎

Theorem Error$DomainCheck

proof)
use *mkerr$declaration*
invoke $\_ \rightarrow \_$
with predicate ($mkerr \in$
$\{f : EXCEPTION \nrightarrow DECISION \mid$
$\forall x : EXCEPTION \bullet \exists y : DECISION \bullet (x, y) \in f\}$)
rewrite
instantiate $x == (message, function)$
invoke *EXCEPTION*
apply *inDom* to predicate
$(message, function) \in \mathrm{dom}\ mkerr$
rearrange
apply *mkerrType*
apply *tupleInCross2* to predicate
$(message, function) \in MESSAGE \times FUNCTION$
simplify
∎

Theorem objSAttsAug_aGrow
$AugmentingAccess \wedge \Xi UsrObj \Rightarrow$
$(\forall ou : UID;\ pa : \mathbb{P}\, PERMISSION \mid ou \in uids \bullet$
$(\forall x : objSAtts\ (\theta ActualPermSet, ou, pa) \bullet$
$x \in objSAtts\ (\theta ActualPermSet', ou, pa))\ )$

proof)
apply *objSAtts_def*1 to expression
$objSAtts(\theta ActualPermSet, ou, pa)$
apply *objSAtts_def*1 to expression
$objSAtts(\theta ActualPermSet', ou, pa)$
with expression (
if $\theta ActualPermSet \in ActualPermSet \wedge$
$ou \in \theta ActualPermSet.uids \wedge$
$pa \in \mathbb{P}\, PERMISSION$
then $\{o\_\_0 : \theta ActualPermSet.oids;\ a\_\_0 : pa \mid a\_\_0 \in$
$\theta ActualPermSet.permSet\ (ou, o\_\_0)$
$\bullet\ o\_\_0\}$
else $objSAtts\ (\theta ActualPermSet, ou, pa)$)
rewrite
invoke $\Xi UsrObj$

apply $UsrObj\$thetasEqual$ to predicate
$\theta UsrObj = \theta UsrObj'$
with expression (
if $\theta ActualPermSet' \in ActualPermSet \land$
$ou \in \theta ActualPermSet'.uids \land$
$pa \in \mathbb{P}\, PERMISSION$
then $\{o\_1 : \theta ActualPermSet'.oids;\ a\_1 : pa \mid a\_1 \in$
$\theta ActualPermSet'.permSet(ou, o\_1)$
$\bullet\ o\_1\}$
else $objSAtts\ (\theta ActualPermSet', ou, pa)$)
rewrite
prove
instantiate $a\_1 == a\_0$
simplify
invoke $AugmentingAccess$
use $augb\_aGrow[f := permSet, g := permSet',$
$up := (u, o), op := (ou, x), na := a]$
rearrange
simplify
apply $subDef$ to predicate
$permSet\ (ou, x) \subseteq permSet'\ (ou, x)$
apply $PERMS\_in\_RIGHTS$ to predicate
$permSet\ (ou, x) \in \mathbb{P}\, RIGHT$
apply $PERMS\_in\_RIGHTS$ to predicate
$permSet'\ (ou, x) \in \mathbb{P}\, RIGHT$
apply $fUOPERMS\_DomApp$ to predicate
$permSet\ (ou, x) \in \mathbb{P}\, PERMISSION$
apply $fUOPERMS\_DomApp$ to predicate
$permSet'\ (ou, x) \in \mathbb{P}\, PERMISSION$
invoke UOp
apply $tupleInCross2$ to predicate
$(ou, x) \in UID \times OID$
apply $tupleInCross2$ to predicate
$(u, o) \in UID \times OID$
with predicate $((ou, x) \in \mathrm{dom}\ permSet)$
rewrite
with predicate $((ou, x) \in \mathrm{dom}\ permSet')$
rewrite
simplify
with predicate $(x \in OID)$
rewrite
instantiate $x\_0 == a\_0$
simplify
∎

Theorem objSAttsAug_na
$AugmentingAccess \land \Xi UsrObj \Rightarrow$
$(\forall ou : UID;\ pa : \mathbb{P}\, PERMISSION \mid ou \in uids \land (u \neq ou \lor a \notin pa) \bullet$
$(\forall x : objSAtts(\theta ActualPermSet', ou, pa) \bullet$
$x \in objSAtts(\theta ActualPermSet, ou, pa)\ )\ )$
proof)

apply *objSAtts_def*1 to expression
objSAtts($\theta ActualPermSet'$, *ou*, *pa*)
apply *objSAtts_def*1 to expression
objSAtts($\theta ActualPermSet$, *ou*, *pa*)
invoke $\Xi UsrObj$
apply *UsrObj*$thetasEqual to predicate
$\theta UsrObj = \theta UsrObj'$
with expression (
if $\theta ActualPermSet' \in ActualPermSet \wedge$
$ou \in \theta ActualPermSet'.uids \wedge$
$pa \in \mathbb{P}\, PERMISSION$
then $\{o\_0 : \theta ActualPermSet'.oids;\ a\_0 : pa \mid a\_0 \in$
$\theta ActualPermSet'.permSet(ou, o\_0)$
$\bullet\ o\_0\}$
else *objSAtts* $(\theta ActualPermSet', ou, pa))$
rewrite
with expression (
if $\theta ActualPermSet \in ActualPermSet \wedge$
$ou \in \theta ActualPermSet.uids \wedge$
$pa \in \mathbb{P}\, PERMISSION$
then $\{o\_1 : \theta ActualPermSet.oids;\ a\_1 : pa \mid a\_1 \in$
$\theta ActualPermSet.permSet(ou, o\_1)$
$\bullet\ o\_1\}$
else *objSAtts* $(\theta ActualPermSet, ou, pa))$
rewrite
prenex
prove
use *augb_naNoDim*$[f := permSet, g := permSet', na := a,$
$up := (u, o), p := (ou, x), a := a\_0]$
apply *notEqRule* to predicate
$((u, o), a) \neq ((ou, x), a\_0)$
apply *tupleInCross*2 to predicate
$((u, o), a) \in (UID \times OID) \times RIGHT$
apply *tupleInCross*2 to predicate
$((ou, x), a\_0) \in (UID \times OID) \times RIGHT$
apply *tupleInCross*2 to predicate
$(u, o) \in UID \times OID$
apply *tupleInCross*2 to predicate
$(ou, x) \in UID \times OID$
rearrange
invoke *UOp*
invoke *AugmentingAccess*
with predicate $((u, o) \in \mathrm{dom}\ permSet)$
rewrite
with predicate $((ou, x) \in \mathrm{dom}\ permSet)$
rewrite
apply *tupleInCross*2 to predicate
$(u, o) \in UID \times OID$
simplify
with predicate $(\neg\ ((u, o), a) = ((ou, x), a\_0))$
rewrite

instantiate $a\_1 == a\_0$
prove by reduce
∎

Theorem objSAttsAug_aDef
$AugmentingAccess \land \Xi UsrObj \Rightarrow$
$(\forall pa : \mathbb{P}\, PERMISSION \mid a \in pa \bullet$
$(\forall x : objSAtts\ (\theta ActualPermSet', u, pa) \bullet$
$x \in objSAtts\ (\theta ActualPermSet, u, pa) \cup \{o\}\ )\ )$
proof)
apply $objSAtts\_def1$ to expression
$objSAtts(\theta ActualPermSet', u, pa)$
apply $objSAtts\_def1$ to expression
$objSAtts(\theta ActualPermSet, u, pa)$
invoke $\Xi UsrObj$
apply $UsrObj\$thetasEqual$ to predicate
$\theta UsrObj = \theta UsrObj'$
invoke $AugmentingAccess$
with expression (
if $\theta ActualPermSet' \in ActualPermSet \land$
$u \in \theta ActualPermSet'.uids \land$
$pa \in \mathbb{P}\, PERMISSION$
then $\{o\_0 : \theta ActualPermSet'.oids;\ a\_0 : pa \mid a\_0 \in$
$\theta ActualPermSet'.permSet(u, o\_0)$
$\bullet\ o\_0\}$
else $objSAtts(\theta ActualPermSet', u, pa)$)
rewrite
with expression (
if $\theta ActualPermSet \in ActualPermSet \land$
$u \in \theta ActualPermSet.uids \land$
$pa \in \mathbb{P}\, PERMISSION$
then $\{o\_1 : \theta ActualPermSet.oids;\ a\_1 : pa \mid a\_1 \in$
$\theta ActualPermSet.permSet(u, o\_1)$
$\bullet\ o\_1\}$
else $objSAtts(\theta ActualPermSet, u, pa)$)
rewrite
with predicate $(x \in \{o\_1 : oids;\ a\_1 : pa \mid$
$a\_1 \in permSet\ (u, o\_1) \bullet o\_1\} \cup \{o\})$
rewrite
with predicate $(x \in \{o\_0 : oids';\ a\_1 : pa \mid$
$a\_1 \in permSet'\ (u, o\_0) \bullet o\_0\})$
rewrite
instantiate $a\_1 == a\_0$
use $augb\_naNoDim[f := permSet, g := permSet', up := (u, o),$
$p := (u, x), na := a, a := a\_0]$
rearrange
invoke UOp
apply $notEqRule$ to predicate
$((u, o), a) \neq ((u, x), a\_0)$
apply $tupleInCross2$ to predicate
$((u, o), a) \in (UID \times OID) \times RIGHT$

apply *tupleInCross2* to predicate
$((u, x), a\_0) \in (UID \times OID) \times RIGHT$
apply *tupleInCross2* to predicate
$(u, o) \in UID \times OID$
apply *tupleInCross2* to predicate
$(u, x) \in UID \times OID$
with predicate $(permSet \in fUO\_PERMS)$
simplify
with predicate $(permSet' \in fUO\_PERMS)$
simplify
reduce
■

Theorem rule T34ii
$AugmentingAccess \wedge$
$\Xi SecClasses \wedge$
$a \in \{Read, Write\} \wedge$
$((u, o), a)$ *secureRel* $\theta SecClasses$
$\Rightarrow$ *PreservesSimpleSecurity*

proof)
invoke *AugmentingAccess*
invoke *PreservesSimpleSecurity*
invoke $\Delta SecClasses$
invoke *SimpleSecureState*
simplify
prenex
split $(p, a\_0) = ((u, o), a)$
cases
prove by reduce
next
use *augb_naNoDim*$[f := permSet, g := permSet', up := (u, o),$
$na := a, a := a\_0]$
rearrange
apply *notEqRule* to predicate
$((u, o), a) \neq (p, a\_0)$
apply *tupleInCross2* to predicate
$((u, o), a) \in (UID \times OID) \times RIGHT$
apply *tupleInCross2* to predicate
$(p, a\_0) \in (UID \times OID) \times RIGHT$
invoke *UOp*
apply *tupleInCross2* to predicate
$(u, o) \in UID \times OID$
simplify
invoke $\Xi SecClasses$
use *SecClasses$thetasEqual*
instantiate $p\_0 == p, a\_1 == a\_0$
simplify
rearrange
prove by reduce
next

∎

Theorem rule T35iii_case1
$AugmentingAccess \land$
$\Xi SecClasses \land$
$a = Read \land$
$(\forall ow : objSAtts\ (\theta ActualPermSet, u, \{ReadWrite, Write\}) \bullet$
$objclass\ ow\ dominates\ objclass\ o)$
$\Rightarrow (\forall w : objSAtts\ (\theta ActualPermSet', u, \{ReadWrite, Write\}) \bullet$
$objclass'\ w\ dominates\ objclass'\ o)$
proof)
instantiate $ow == w$
use $objSAttsAug\_na[ou := u,$
$pa := \{Write, ReadWrite\},$
$x := w]$
rearrange
with predicate $(AugmentingAccess \land$
$\Xi UsrObj \land$
$u \in UID \land$
$u \in uids \land$
$w \in objSAtts(\theta ActualPermSet', u,$
$(\{Write\} \cup \{ReadWrite\})) \land$
$\{Write\} \cup \{ReadWrite\} \in \mathbb{P}\ PERMISSION \land$
$(u \neq u \lor a \notin \{Write\} \cup \{ReadWrite\})$
reduce
invoke $\Xi SecClasses$
apply $SecClasses\$thetasEqual$ to predicate
$\theta SecClasses = \theta SecClasses'$
reduce
∎

Theorem rule T35iii_case2
$AugmentingAccess \land StarPropertyState \land \Xi SecClasses \land a = Read$
$\Rightarrow (\forall w : objSAtts\ (\theta ActualPermSet', u, \{ReadWrite, Write\});$
$r : objSAtts\ (\theta ActualPermSet', u, \{ReadWrite, Read\}) \mid$
$\neg\ r = o \bullet objclass'\ w\ dominates\ objclass'\ r)$
proof)
invoke $StarPropertyState$
instantiate $u\_0 == u, ow == w, or == r$
use $objSAttsAug\_na[$
$pa := \{ReadWrite, Write\}, ou := u, x := w]$
use $objSAttsAug\_aDef[$
$pa := \{ReadWrite, Read\}, x := r]$
rearrange
with predicate $(a \in \{ReadWrite\} \cup \{Read\})$
rewrite
with predicate $(\{ReadWrite\} \cup \{Read\} \in \mathbb{P}\ PERMISSION)$
reduce
with predicate $(\{ReadWrite\} \cup \{Write\} \in \mathbb{P}\ PERMISSION)$
reduce

with predicate $(u \neq u \vee a \notin \{ReadWrite\} \cup \{Write\})$
reduce
simplify
invoke $\Xi SecClasses$
apply $SecClasses\$thetasEqual$ to predicate
$\theta SecClasses = \theta SecClasses'$
simplify
invoke $AugmentingAccess$
reduce
■

Theorem rule T35iii_case3
$AugmentingAccess \wedge StarPropertyState \wedge \Xi SecClasses \wedge a = Read$
$\Rightarrow (\forall ou : uids \mid \neg ou = u \bullet$
$(\forall w : objSAtts\ (\theta ActualPermSet', ou, \{ReadWrite, Write\});$
$r : objSAtts\ (\theta ActualPermSet', ou, \{ReadWrite, Read\}) \bullet$
$objclass'\ w\ dominates\ objclass'\ r))$

proof)
use $objSAttsAug\_na$
$[pa := \{ReadWrite, Write\}, x := w]$
use $objSAttsAug\_na$
$[pa := \{ReadWrite, Read\}, x := r]$
with predicate $(\{ReadWrite\} \cup \{Write\} \in \mathbb{P}\ PERMISSION)$
reduce
with predicate $(\{ReadWrite\} \cup \{Read\} \in \mathbb{P}\ PERMISSION)$
reduce
rearrange
with predicate $(u \neq ou \vee a \notin \{ReadWrite\} \cup \{Read\})$
rewrite
with predicate $(u \neq ou \vee a \notin \{ReadWrite\} \cup \{Write\})$
rewrite
invoke $\Xi SecClasses$
apply $SecClasses\$thetasEqual$ to predicate
$\theta SecClasses = \theta SecClasses'$
simplify
with predicate $(\Xi UsrObj \wedge ou \in UID)$
rewrite
with predicate $(\Xi UsrObj \Rightarrow r \in objSAtts(\theta ActualPermSet, ou,$
$(\{ReadWrite\} \cup \{Read\})))$
reduce
with predicate $(\Xi UsrObj \Rightarrow w \in objSAtts(\theta ActualPermSet, ou,$
$(\{ReadWrite\} \cup \{Write\})))$
reduce
invoke $StarPropertyState$
instantiate $u\_\_0 == ou, ow == w, or == r$
rearrange

with predicate ($ou \in UID \land$
$w \in OID \land$
$r \in OID \land$
$ou \in uids \land$
$w \in objSAtts(\theta ActualPermSet, ou,$
$(\{Write\} \cup \{ReadWrite\})) \land$
$r \in objSAtts(\theta ActualPermSet, ou,$
$(\{Read\} \cup \{ReadWrite\}))$)
reduce
simplify
■

Theorem rule T35iii
$AugmentingAccess$
$\land \Xi SecClasses$
$\land a = Read$
$\land (\forall ow : objSAtts\ (\theta ActualPermSet, u, \{ReadWrite, Write\}) \bullet$
$objclass\ ow\ dominates\ objclass\ o)$
$\Rightarrow PreservesStarProperty$

proof)
invoke $PreservesStarProperty$
invoke $\Delta SecClasses$
invoke $StarPropertyState$
simplify
prenex
split $u\_0 = u \land or = o$
cases
use $T35iii\_case1[w := ow]$
rearrange
reduce
next
split $u\_0 = u$
cases
simplify
use $T35iii\_case2[r := or, w := ow]$
rearrange
simplify
invoke $StarPropertyState$
simplify
reduce
next
simplify
use $T35iii\_case3[r := or, w := ow, ou := u\_0]$
rearrange
simplify
reduce
next
■

Theorem GetReadOkYes$DomainCheck

proof)
cases
invoke *ACLS*
prove
next
apply *pfunAppliesTo* to predicate
$(usrclass, u?) \in applies\$to$
apply *pfunAppliesTo* to predicate
$(objclass, o?) \in applies\$to$
invoke *SecClasses*
invoke *fUC*
invoke *fOC*
prove
next
apply *objSAtts_dom* to expression
dom *objSAtts*
cases
prove by reduce
next
prove
apply *pfunAppliesTo* to predicate
$(objclass, o) \in applies\$to$
apply *pfunAppliesTo* to predicate
$(objclass, o?) \in applies\$to$
invoke *SecClasses*
invoke *fOC*
prove
apply *objSAtts_def* 1 to expression
$objSAtts\ (\theta ActualPermSet[oids := \text{dom}\ objclass,$
$uids := \text{dom}\ usrclass],$
$u?, (\{Write\} \cup \{ReadWrite\}))$
prove
with predicate ($Write \in PERMISSION \wedge$
$ReadWrite \in PERMISSION$)
reduce
simplify
next
∎

Theorem GetReadOkNo$DomainCheck

proof)
invoke *ACLS*
cases
prove
next
prove by reduce
next
cases
apply *objSAtts_dom* to expression
dom *objSAtts*

with predicate $((\theta ActualPermSet, u?, \{Write\} \cup \{ReadWrite\})$
$\in \{m : ActualPermSet;\ u : UID;$
$atts : \mathbb{P}\, PERMISSION \mid$
$u \in m.uids\})$
rewrite
with predicate $(\{Write\} \cup \{ReadWrite\} \in \mathbb{P}\, PERMISSION)$
reduce
simplify
next
prenex
apply $objSAtts\_def1$ to expression
$objSAtts\ (\theta ActualPermSet, u?,$
$(\{Write\} \cup \{ReadWrite\}))$
prove
with predicate $(Write \in PERMISSION \land$
$ReadWrite \in PERMISSION)$
reduce
reduce
next
■

Theorem GetReadRejectIsSound
$\forall\, GetReadReject \bullet$
$GetReadNotAffected \Rightarrow PreservesSimpleSecurity \land$
$PreservesStarProperty$

proof)
apply $SimpleSecurityNotAffected$
apply $StarPropertyNotAffected$
prove
■

Theorem GetReadOkYesIsSound
$\forall\, GetReadOkYes \bullet$
$GetReadNotAffected \Rightarrow PreservesSimpleSecurity \land$
$PreservesStarProperty$

proof)
cases
invoke $GetReadOkYes$
invoke $GetReadNotAffected$
use $T34ii[o?/o, u?/u, a := Read]$
apply $secureRel\_Def$ to predicate
$((u?, o?), Read)secureRel\theta SecClasses$
with predicate $(\theta SecClasses \in SecClasses$
$\land\ u? \in \theta SecClasses.uids$
$\land\ o? \in \theta SecClasses.oids$
$\land\ Read \in PERMISSION)$
reduce
with predicate $(Read \in \{Read\} \cup \{ReadWrite\})$
reduce

with predicate (
$\theta SecClasses.usrclass\ u?\ dominates\ \theta SecClasses.objclass\ o?$)
reduce
with predicate ($Read \in \{Read\} \cup \{Write\}$)
reduce
simplify
next
use $T35iii[o?/o, u?/u, a := Read]$
rearrange
simplify
invoke *GetReadOkYes*
with predicate ($\forall ow : objSAtts(\theta ActualPermSet, u?,$
$(\{ReadWrite\} \cup \{Write\}))$
$\bullet\ objclass\ ow\ dominates\ objclass\ o?$)
rewrite
simplify
next
■

Theorem GetReadIsSound
$\forall GetRead \bullet PreservesSimpleSecurity \wedge PreservesStarProperty$

proof)
invoke *GetRead*
split *GetReadOkYes*
cases
use *GetReadOkYesIsSound*
simplify
next
use *GetReadRejectIsSound*
simplify
next
■

Bibliografía

[1] Marschall D. Abrams, Sushil Jajodia, and Harold J. Podell, *Information security: an integrated collection of essays.*

[2] Ross J. Anderson, *Introduction to security.*

[3] ———, *Multilevel security systems.*

[4] Konstantin Beznosov and Yi Deng, *Engineering access control in distributed applications.*

[5] Morrie Gasser, *Building a secure computer system*, Van Nostrand Neinhold, New York, 1988.

[6] R.K. Gupta, *Exploring the world of application servers*, (2002).

[7] Jacky, *The way of Z*, Cambridge University Press, 1997.

[8] Leonard J. LaPadula and D. Elliott Bell, *Secure computer systems: A mathematical model*, MITRE Technical Report 2547 **II** (1973).

[9] ———, *Secure computer systems: Mathematical foundations*, MITRE Technical Report 2547 **I** (1973).

[10] Peter A. Loscocco, Stephen D. Smalley, Patrick A. Muckelbauer, Ruth C. Taylor, S. Jeff Turner, and John F. Farrel, *The inevitability of failure: The flawed assumption of security in modern computing environments.*

[11] J. McLean, *The specification and modeling of computer security*, IEEE Computer, 1990.

[12] Department of Defense, *Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD.

[13] Mark Saaltink, *The Z/EVES 2.0 user's guide.*

[14] T. Sundsted, *Application servers: An introduction*, (2001).

[15] Jim Woodcock and Jim Davies, *Using Z: Specification, Refinement, and Proof*, Prentice Hall International Series in Computer Science, 1996.

[16] www.sun.com, *Secure computing with Java: Now and the future.*